

# **CERES Conversion Guide**

**By**

**James L. Donaldson**

**Raytheon Technical Services Company  
130 Research Drive  
Hampton, Virginia 23666**

**January 18, 2005**



Table of Tables.....	iv
Table of Figures .....	v
Table of Listings .....	vi
Acronyms and Abbreviations.....	vii
1 Introduction.....	1
2 Objective .....	1
3 The Macintosh G5 Target Environment .....	2
3.1 What is Darwin .....	2
3.2 The Unix Command Line .....	3
3.3 Other Useful Tools.....	3
3.3.1 The gcc Compiler Group and Xcode .....	4
3.3.2 TextEdit.....	6
3.3.3 The Activity Monitor .....	7
3.4 The Console Utility.....	11
3.5 The IBM XL FORTRAN Compiler (IBM XLF).....	14
3.5.1 What Standard Should I Use? .....	14
3.5.2 Initial Build and Language Level Assessment .....	14
3.5.3 Complete Build and Sweep for Array Bounds Violations .....	19
3.5.4 Complete Build and Sweep for Invalid Floating-Point Operations .....	20
3.5.5 Complete Build for Safe Optimization and Results Comparison.....	21
3.6 The Darwin Static Linker (ld) and Archive Library (ar) Tools .....	22
3.7 The Make Utility.....	22
4 A Process for Porting CERES Science Code.....	22
4.1 Plan Your Work .....	22
4.1.1 Document the Subsystem Dependencies .....	25
4.1.2 Document the Subsystem Components .....	27
4.1.3 Document the Subsystem Directory Model.....	27
4.1.4 Document the Target Platform.....	28
4.1.4.1 The Endian Thing.....	28
4.1.4.2 Source OS to Target OS.....	28
4.1.4.3 IEEE Compliance.....	28
4.1.4.4 Compiler Availability and Compatibility .....	29
4.1.5 Document Your High Level Conversion Strategy .....	29
4.1.6 Document Your Detailed Conversion Plan.....	30
4.1.7 Estimate and Document Your Subsystem Conversion Schedule .....	32
4.2 Work Your Plan.....	32
4.3 Library Install Example .....	33
4.3.1 Create a Delivery Package on the Source Platform.....	34
4.3.2 Install the Subsystem Source on the Target Platform.....	35
4.3.3 Build the Library.....	36
4.4 Build PGE and Test Example .....	43
4.4.1 Build the PGE .....	43
4.4.2 Test the PGE .....	44
4.5 Tracking Down an Invalid Floating Point Operation .....	49
4.6 Comparing Results with the Benchmark .....	54
4.6.1 Learning to Live with NaN's and INF's .....	55

## CERES Conversion Guide

4.6.2 Mixed Mode Arithmetic .....	55
4.6.3 Divide by Zero .....	57
4.6.4 Proper Usage of Intrinsic Functions .....	60
4.6.5 Array Boundary Violations .....	61
4.6.6 Un-initialized Local Variables.....	62
4.6.7 Executing Unnecessary Code.....	63
5 A Conversion Checklist .....	66
6 Findings from the SARB Conversion Effort .....	67
6.1 The PGS Toolkit Installation on the Mac G5 .....	67
6.1.1 Installing zlib.....	67
6.1.2 Installing JPEG .....	67
6.1.3 Installing HDF4 .....	68
6.1.4 Installing HDF5 .....	68
6.1.5 Installing HDF-EOS Version 2 .....	68
6.1.6 Installing HDF-EOS Version 5 .....	69
6.1.7 Installing the Toolkit.....	69
6.1.7.1 IBM XLF and Not g77 .....	69
6.1.7.2 Searching for search.h.....	69
6.1.7.3 I Can't Find zlib and JPEG! .....	69
6.1.7.4 Installing the Toolkit Ancillary/Auxiliary Data Access Tools .....	69
6.1.7.5 Where is malloc.h? .....	70
6.1.7.6 The Case of the Trailing Underscore .....	70
6.1.7.7 Toolkit User Accounts .....	70
6.2 The CERES Library Installation on the Mac G5 .....	71
6.2.1 Installing the CERES Libraries.....	71
6.2.1.1 C Files Can't Find malloc.h.....	72
6.2.1.2 Syntax Problem with ALLOCATABLE Array Declarations .....	73
6.2.2 Installing the CERES Library Test Suite .....	73
6.2.2.1 SGI-specific C Compiler Test Removed .....	73
6.2.2.2 SGI-specific Compiler Defaults Test Removed .....	73
6.2.2.3 Unsatisfied External in Pcf_c Test.....	74
6.2.2.4 Make File Bug Corrected .....	74
6.2.3 Testing the CERES Libraries with the Test Suite.....	74
6.2.3.1 We Agree to Differ .....	74
6.2.3.2 Legitimate Test Suite Errors .....	76
6.2.4 Testing CERESlib While Testing the SARB Subsystem .....	76
6.2.4.1 The CERES Validation Regions Problem .....	77
6.2.4.2 The Case of the Missing Array Index.....	77
6.3 The SARB Installation on Mac G5 .....	78
6.3.1 SARB Library and PGE Compilations .....	78
6.3.2 SARB Runtime Issues.....	80
6.3.2.1 SARB Issues with Mixed Mode Arithmetic .....	80
6.3.2.2 SARB Issues with Divide by Zero .....	82
6.3.2.3 SARB Issues with Intrinsic Functions .....	82
6.3.2.4 SARB Issues with Array Boundary Violations .....	83
6.3.2.5 SARB Issues with Un-initialized Local Variables.....	84

6.3.2.6 SARB Issues with Executing Unnecessary Code .....	84
6.3.2.7 SARB Mystery Anomaly .....	85
6.3.3 SARB Verification Results on Mac G5 .....	86
6.3.3.1 Verification Results for the SARB Monthly Preprocessors .....	86
6.3.3.2 Verification Results for the SARB Main Processor .....	88
6.3.3.3 Verification Results for the SARB Postprocessor .....	91
6.3.3.4 Verification Results for the SARB QC Summary Processor .....	92
7 Lessons Learned From the SARB Conversion.....	99
7.1 Adopt a Coding Standard and be Consistent .....	99
7.1.1 Comply with the FORTRAN Standard .....	100
7.1.2 Follow Good Programming Practices .....	101
7.2 Use Root Sparingly .....	104
7.3 Develop a Contingency Plan.....	104
7.4 Eliminate Invalid Floating Point Operations .....	106
7.5 Eliminate Array Boundary Violations .....	106
7.6 Eliminate Mixed Mode Arithmetic .....	107
7.7 Consider Developing a Test Suite .....	107
7.8 Do Not Abuse the -qsave Compiler Switch .....	108
7.9 DATA Statements in Module Headers .....	108
8 Writing Optimal Code.....	110
8.1 Comment Your Code .....	110
8.2 Source-Level Optimization.....	112
8.2.1 Use Array Notation Instead of Pointers .....	112
8.2.2 Unrolling Small Loops.....	112
8.2.3 Long Logical IF Expressions .....	113
8.2.4 Arrange Boolean Operands for Quick Expression Evaluation .....	113
8.2.5 Unnecessary Store-to-Load Dependencies .....	113
8.2.6 Arranging Cases by Probability of Occurrence .....	114
8.2.7 Generic Loop Hoisting.....	115
8.2.8 Sorting and Padding User Defined Types.....	115
9 Writing Production Code .....	116
10 Summary .....	117

## Table of Tables

Table 1 - IBM XLF Environment Variable Settings .....	15
Table 2 - IBM XL Fortran Compiler Language Level Suboptions .....	19
Table 3 - Mac G5 Versus SGI Mixed Mode Results .....	56
Table 4 - Mac G5 Vs SGI Following Mods.....	57
Table 5 - SARBlib Compiler Warning Messages .....	79
Table 6 - SARBlib Fatal Compiler Errors .....	79
Table 7 - SARB Main Processor Compiler Warning/Error .....	80
Table 8 - SARB QC Summary Processor Compiler Warnings .....	80
Table 9 - SARB Runtime Issues with Mixed Mode Arithmetic .....	81
Table 10 - SARB Divide by Zero Events .....	82
Table 11 - SARB Invalid Intrinsic Function Arguments .....	83
Table 12 - SARB Array Bounds Violations .....	84
Table 13 - SARB Issues with Executing Unnecessary Code.....	85
Table 14 - SARB Mystery Anomaly .....	85
Table 15 - SARB Monthly Preprocessor Test Case ID .....	87
Table 16 - CER5.1P1 Test Case Parameters.....	88
Table 17 - CER5.4P1 Output Files and Formats .....	92

## Table of Figures

Figure 1 Terminal Preferences Dialog.....	3
Figure 2 The Finder Window Showing the Applications Directory.....	4
Figure 3 Finder View Using Columns .....	5
Figure 4 Macintosh Display After Invoking TextEdit.....	6
Figure 5 Macintosh Window Control Buttons .....	6
Figure 6 Trace-back Report in Terminal Window.....	7
Figure 7 The Activity Manager Window .....	8
Figure 8 Floating CPU Monitor.....	8
Figure 9 InstSARB_Drv.exe Open Files .....	9
Figure 10 InstSARB_Drv.exe Process Statistics .....	9
Figure 11 InstSARB_Drv.exe Memory Utilization.....	10
Figure 12 Console Utility Window .....	11
Figure 13 Log Selection Controls .....	12
Figure 14 InstSARB_Drv.exe.crash.log Window.....	13
Figure 15 – Sample Process Diagram.....	30
Figure 16 – Object Listing Header for Subroutine CldLyr_ID.....	46
Figure 17 – Object Address of Fault .....	47
Figure 18 – Using Xcode to Find and View Source Line .....	48
Figure 19 - Terminal Window Traceback.....	49
Figure 20 - Object Header Line for Subroutine tune_xxx .....	51
Figure 21 - Object Address of Invalid Floating Point Operation.....	52
Figure 22 - Offending Source Line in Tune_Code.f90 .....	53

## Table of Listings

Listing 1 - Outline of SARB Conversion Plan.....	25
Listing 2 - Excerpt from SARB Conversion Plan.....	31
Listing 3 - CERES Environment Variable Definition Script.....	37
Listing 4 - SARBlib Makefile Script .....	40
Listing 5 - Build Script for Status Message Files .....	42
Listing 6 - CrashReporter Log .....	45
Listing 7 - Terminal Window Diagnostic Output .....	48
Listing 8 - CrashReporter Log for SR tune_xxx.....	50
Listing 9 - Code Excerpt for Divide by Zero Problem.....	58
Listing 10 - Code Excerpt With Workaround for Divide by Zero .....	59
Listing 11 - Code Excerpt for LOG of Zero .....	60
Listing 12 - Code Excerpt with Workaround for LOG of Zero .....	60
Listing 13 - CrashReporter Log for Array Bounds Violation.....	61
Listing 14 - CrashReporter Log for Unnecessary Code Example.....	64
Listing 15 - Conditional Compilation for malloc .....	72
Listing 16 - IBM XLF Syntax Errors.....	73
Listing 17 - Revised Syntax for ALLOCATABLE Arrays .....	73
Listing 18 - Make File Typographical Error .....	74
Listing 19 - Acceptable Comparison Mismatch .....	75
Listing 20 - Acceptable Comparison Differences.....	75
Listing 21 - More Acceptable Comparison Differences .....	76
Listing 22 - Missing Array Index.....	77
Listing 23 - CER5.0P1 Comparison Results for 200110 .....	87
Listing 24 - CER5.1P1 Comparison Mismatches with Filtered BTCF's.....	89
Listing 25 - CER5.1P1 Excerpt of Comparison Output for QC Report .....	90
Listing 26 - CER5.1P1 Typical Entry in QC Report Comparison Output .....	91
Listing 27 - HDF Verification Results Via E-mail .....	92
Listing 28 - Monthly CERES Region Report .....	96
Listing 29 - Monthly CERES CRS Hour Availability Table.....	97
Listing 30 - Tuning Error Statistics Anomaly.....	98
Listing 31 - DATA Statement Excerpt .....	100
Listing 32 - Code Excerpt with Computed Array Indices .....	102
Listing 33 - Suggested Fix for Good Programming Practice.....	103
Listing 34 - Code Excerpt With No Comments .....	111
Listing 35 - grep Results for 3.1415 .....	112
Listing 36 - Example for Loop Unrolling .....	113
Listing 37 - Results of Loop Unroll .....	113
Listing 38 - Store-to-Load Dependency.....	114
Listing 39 - Avoiding Store-to-Load Dependency .....	114
Listing 40 - CASE Statement Not in Most Probable Order .....	114
Listing 41 - CASE Statement in Most Probable Order .....	115
Listing 42 - Redundant Constant Evaluation Code.....	115
Listing 43 - Reduction of Loop-invariant Constant Expression.....	115



## Acronyms and Abbreviations

<i>Acronym</i>	<i>Description</i>
<b>.a</b>	Library archive file
<b>.csh</b>	C Shell script file
<b>.exe</b>	FORTTRAN executable file
<b>f</b>	FORTTRAN-77 source file
<b>f90</b>	FORTTRAN-90 source file
<b>.gz</b>	GNU zip file
<b>.h</b>	C header file
<b>.log</b>	Log file
<b>.met</b>	Metadata file
<b>.mod</b>	FORTTRAN Module file
<b>.o</b>	FORTTRAN and C object file
<b>.t</b>	Status Message file
<b>.txt</b>	ASCII text file
<b>.Z</b>	Compressed file
<b>5.0P1</b>	SARB Pre-processor PGE
<b>5.1P1</b>	SARB Main Processor PGE
<b>5.3P1</b>	SARB Post-processor PGE
<b>5.4P1</b>	SARB QC Post-processor
<b>AA</b>	Ancillary/Auxiliary
<b>API</b>	Application Interface
<b>ASCII</b>	American Standard Code for Information Interchange
<b>ASDC</b>	Atmospheric Sciences Data Center
<b>CC</b>	Configuration Code
<b>cd</b>	Unix case sensitive change directory command
<b>CER</b>	CERES
<b>CERES</b>	Clouds and the Earth's Radiant Energy System
<b>CERESlib</b>	CERES library
<b>cp</b>	Unix case sensitive copy command
<b>CPU</b>	Central Processing Unit
<b>CRS</b>	Cloud Radiative Swath
<b>CRSB</b>	Cloud Radiative Swath Binary
<b>CRSVB</b>	Cloud Radiative Swath Validation Binary
<b>DAAC</b>	Distributed Active Archive Center
<b>DEC</b>	Digital Equipment Corporation
<b>ECS</b>	EOSDIS Core System
<b>EOS</b>	Earth Observing System
<b>EOSDIS</b>	EOS Data Information System
<b>EOS-AM</b>	EOS Morning Crossing Mission AKA Terra
<b>EOS-PM</b>	EOS Afternoon Crossing Mission AKA Aqua
<b>F77</b>	FORTTRAN-77
<b>F90</b>	FORTTRAN-90
<b>F95</b>	FORTTRAN-95

## CERES Conversion Guide

<b><i>FMI</i></b>	Flight Model 1 (Terra)
<b><i>FM2</i></b>	Flight Model 2 (Terra)
<b><i>FM3</i></b>	Flight Model 3 (Aqua)
<b><i>FM4</i></b>	Flight Model 4 (Aqua)
<b><i>FTP</i></b>	File Transfer Protocol
<b><i>GSFC</i></b>	Goddard Space Flight Center
<b><i>GNU</i></b>	GNU's Not Unix (recursively defined acronym)
<b><i>HDF</i></b>	Hierarchical Data Format
<b><i>HMAER</i></b>	MODIS Aerosol History
<b><i>HMAVAIL</i></b>	Monthly History Available (QC report)
<b><i>HMPSAL</i></b>	Monthly Surface Albedo History
<b><i>HP</i></b>	Hewlett Packard
<b><i>IEEE</i></b>	Institute of Electrical and Electronic Engineers
<b><i>JPEG</i></b>	Joint Photographic Experts Group
<b><i>LaRC</i></b>	Langley Research Center
<b><i>Mac</i></b>	Macintosh
<b><i>MCF</i></b>	Metadata Control File
<b><i>mkdir</i></b>	Unix case sensitive create directory command
<b><i>MOA</i></b>	Meteorological, Ozone, and Aerosol Ancillary
<b><i>MODIS</i></b>	Moderate Resolution Imaging Spectroradiometer
<b><i>mv</i></b>	Case sensitive Unix move command
<b><i>NASA</i></b>	National Aeronautics and Space Administration
<b><i>NCSA</i></b>	National Center for Supercomputing Applications
<b><i>NOAA</i></b>	National Oceanic and Atmospheric Administration
<b><i>OS</i></b>	Operating System
<b><i>PC</i></b>	Personal Computer
<b><i>PCF</i></b>	Process Control File
<b><i>PGE</i></b>	Product Generation Executive
<b><i>PGS</i></b>	Product Generation System
<b><i>PI</i></b>	Principal Investigator
<b><i>PPC</i></b>	Power PC
<b><i>PS</i></b>	Production Strategy
<b><i>QC</i></b>	Quality Control
<b><i>RCF</i></b>	Resource Control File
<b><i>SAH</i></b>	Surface Albedo History
<b><i>SARB</i></b>	Surface and Atmospheric Radiation Budget
<b><i>SARBlib</i></b>	SARB library
<b><i>SCCR</i></b>	System Configuration Change Request
<b><i>SCF</i></b>	Science Computing Facility
<b><i>SDP</i></b>	Science Data Processing
<b><i>SDPTK</i></b>	Science Data Processing Toolkit
<b><i>setenv</i></b>	C Shell set environment variable command (case sensitive)
<b><i>SMF</i></b>	Status Message Facility
<b><i>SS</i></b>	Sampling Strategy
<b><i>SSF</i></b>	Single Satellite Footprint
<b><i>SSFA</i></b>	Single Satellite Footprint Supplemental Aerosol

## CERES Conversion Guide

<b><i>SSFB</i></b>	Single Satellite Footprint Binary
<b><i>su</i></b>	Unix case sensitive switch user command
<b><i>SW</i></b>	Software
<b><i>tar</i></b>	Case sensitive Unix tape archive command
<b><i>TBD</i></b>	To Be Determined
<b><i>TRMM</i></b>	Tropical Rainfall Measuring Mission
<b><i>US</i></b>	United States

---

## 1 Introduction

The Clouds and Earth's Radiant Energy System (CERES) is a key component of the Earth Observing System (EOS) program and is designed to provide a consistent data base of accurately known fields of radiation and clouds to increase our understanding of the Earth as a system. The EOS Program is driven by two major objectives: (1) to acquire essential, global Earth science data on a long term basis; and (2) to provide the Earth science research community with efficient and reliable access to a complete set of data from US and international platforms through the EOS Data and Information System (EOSDIS). The CERES experiment contributes significantly to both objectives. The first is fulfilled by the CERES measurements of the global distribution of the energy input to and the energy output from the Earth. CERES satisfies the second objective by providing data products from the three separate platforms within the EOSDIS environment. The data products include CERES Scanner Data, Instantaneous Clouds and Radiation Data, Synoptic Clouds and Radiation, and Daily and Monthly Averaged Data. CERES provides data required to understand the relative importance of different cloud processes and their interaction with the Earth's climate. These data allow definition of trends in the clear-sky fluxes and the impact of clouds on the Earth's climate and radiation budget. They will also be fundamental data for experiments in long-range weather forecasting and climate prediction.

There are currently 13 different subsystems in the CERES Science Data Processing environment, each composed of one or more Product Generation Executables (PGE) and related control codes in the form of Process Control Files (PCF). There are 11 data products produced by CERES and distributed to the general public.

Code development and testing takes place in the CERES Science Computing Facility, located in Building 1250 with data product production and distribution to the public occurring at the Langley Research Center (LaRC) Distributed Active Archive Center (DAAC) located in Building 1268. Codes are largely built in FORTRAN 90 with some using ADA, IDL with control scripts built in PERL, and UNIX shell scripting language. Currently, production is performed in an SGI 3800 environment. Initial processing occurs subsequent to all data inputs being received with subsystems being exercised in a specific order to generate the successively higher order data products. Re-processing of the data as algorithms are improved is expected to occur several more times over the life of the data. To capture the scientific value of a new algorithm, the entire collection of observed data must be re-processed within a year from initiation of a production run. This requires processing 10 data months of data per processing month, and is known as "10x" processing. Costs associated with the SGI computational environment are quite high and constrain this reprocessing of data with improved algorithms. It is anticipated that running in an open source environment will permit more and faster re-processing with the same high quality results.

## 2 Objective

A pilot project was conducted with the CERES subsystem, Surface and Atmospheric Radiation Budget (SARB). The SARB pilot project involved porting SARB to a Macintosh G5 host, and to the commodity-based, open source cluster implemented by the

DAAC<sup>1</sup>. The primary objective of the pilot project was to demonstrate the feasibility of obtaining “10x” production processing for SARB using the commodity-based open source cluster whose nodes incorporate dual Power PC 970 central processing units running under the Linux operating system.<sup>2</sup> The secondary objective of the pilot project was to establish a process for standardizing the conversion of the rest of the CERES subsystems. This document describes the key lessons learned from the experience gained during the SARB port, and this document describes an approach to the generic process of porting the rest of the subsystems.

### 3 The Macintosh G5 Target Environment

Everything in this section is based on the assumption that porting subsystem code from the Mac G5 to the Linux-based cluster at the DAAC is a problem-free software build followed by a verification using existing test suite software. This assumption is dependent on the fact that the G5 and the cluster nodes are the same hardware architecture, and the requirement that the same compiler be used on the G5 and the DAAC cluster. The SARB conversion experience verifies that performing a Unix to Unix conversion between dissimilar computer architectures is far more difficult than performing a Unix to Linux conversion between identical CPU architectures using the same compiler.

#### 3.1 What is Darwin

This document was written when the current operating system for the Mac G5 was OS X Panther version 10.3.6. The Panther OS is a windowed architecture built on top of a complete Unix implementation named Darwin. If you are at the console of a Mac G5 (see Figure 2), a Darwin command-line terminal window can be opened by clicking on the ‘Applications’ folder<sup>3</sup>, followed by clicking on the ‘Utilities’ folder. In the ‘Utilities’ folder you will find an icon labeled ‘Terminal’. Double clicking the ‘Terminal’ icon will open a Unix command line window<sup>4</sup> similar to an ‘xterm’ window on a more conventional Unix platform (see Figure 6). Starting with Panther 10.3.3, the default shell in the ‘Terminal’ window is “bash” or Bourne Again Shell. If you prefer “csh” (C-Shell) or “tcsh” (Extended C-Shell), it is an easy change to make<sup>5</sup>. Click once on the ‘Terminal’ icon to activate the ‘Terminal’ menu bar at the top of the display. In the menu bar, click ‘Terminal’ and then select ‘Preferences’. You will see a dialog box (see Figure 1) with the title, “Terminal Preferences”. In the terminal preferences dialog box, select the radio button that says, “Execute this command (specify complete path)”. In the text entry area

---

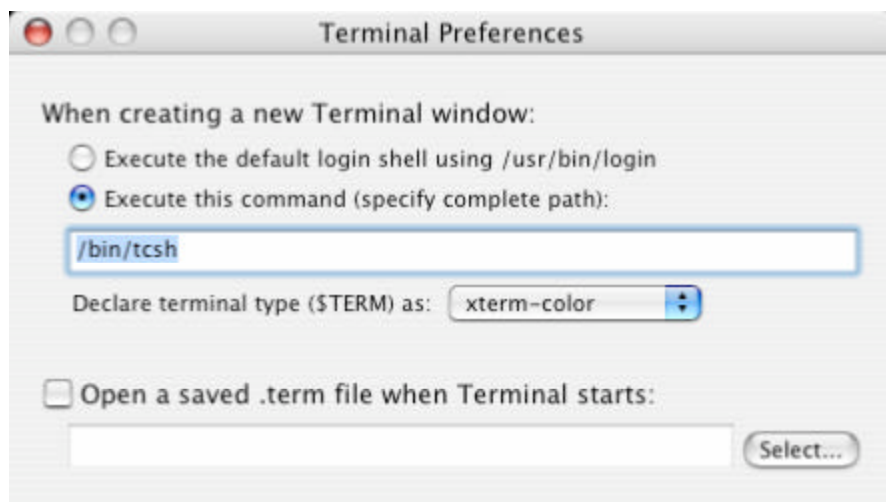
<sup>1</sup> The Mac G5 development platform incorporates dual PPC970 CPU’s that can be exploited via Darwin, an implementation of Berkeley Software Design (BSD) Unix. The Mac G5 operating system, OS X, is built on top of Darwin and it provides a host of helpful development tools to augment the command-line approach to CERES code maintenance.

<sup>2</sup> At this writing the DAAC cluster is still not available for test and timing studies. However, the Mac G5 execution times were very favorable in comparison with their counterparts on the SGI platform.

<sup>3</sup> Use the Finder application to display the several top-level folders that are accessible via the windowed user interface for the Mac. The Finder is always active and is the portal of the Mac OS X operating system. If no Finder window is open, one can be opened by clicking on the smiling face icon on the Dock.

<sup>4</sup> This action also puts the Terminal application on the “Dock”. To open more terminal windows, click and hold with the cursor over the Terminal icon on the Dock. A pop-up menu will appear allowing you to open a new shell (terminal window).

<sup>5</sup> This guide is not a tutorial on Darwin or Unix, but just this once.



**Figure 1 Terminal Preferences Dialog**

just below the radio button, enter the complete path for the shell you want to use every time you open a Terminal window (shell). For example, to use the tcsh shell, enter “/bin/tcsh” without the quotes. For the csh shell, enter, “/bin/csh” without the quotes. To get a terminal window open with the new shell may require you to completely close all the terminal windows that are currently open before you can open a terminal window with the shell that you selected<sup>6</sup>.

### **3.2 The Unix Command Line**

When you open a terminal window (see Figure 6), you are opening a shell. You are now able to execute the same Unix commands and shell scripts that you were using on the SGI platforms like Thunder.<sup>7</sup>

### **3.3 Other Useful Tools**

The Panther OS is supplied with the latest version of the gcc compiler group and an integrated development environment called Xtools. Both gcc and the Xtools are required by the IBM XLF compiler for software application development. This guide is based on experience gained with the IBM XLF compiler, so it is given that the IBM XLF compiler is the primary means of compiling development and production FORTRAN code. A general purpose text editor named TextEdit comes bundled with Panther, and this editor is quite useful for copying and pasting text from one application to another. Another useful tool for monitoring process statistics and for managing processes is the Activity Monitor that also comes bundled with the Macintosh OS.

<sup>6</sup> Every time you open a terminal window, depending on what shell you are using, a .login or .cshrc file will be executed if they exist. You hide these files in your home directory just like any other Unix system.

<sup>7</sup> Well, almost. There are some differences and some of them will be covered in this document.

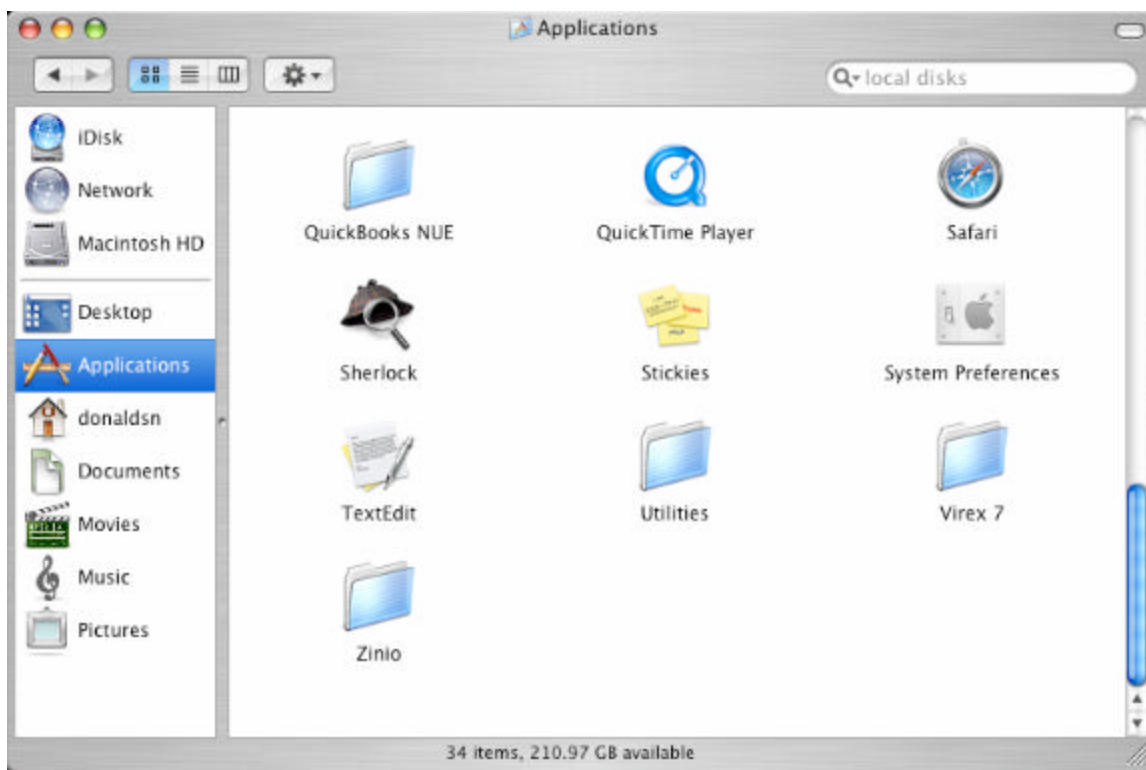


Figure 2 The Finder Window Showing the Applications Directory

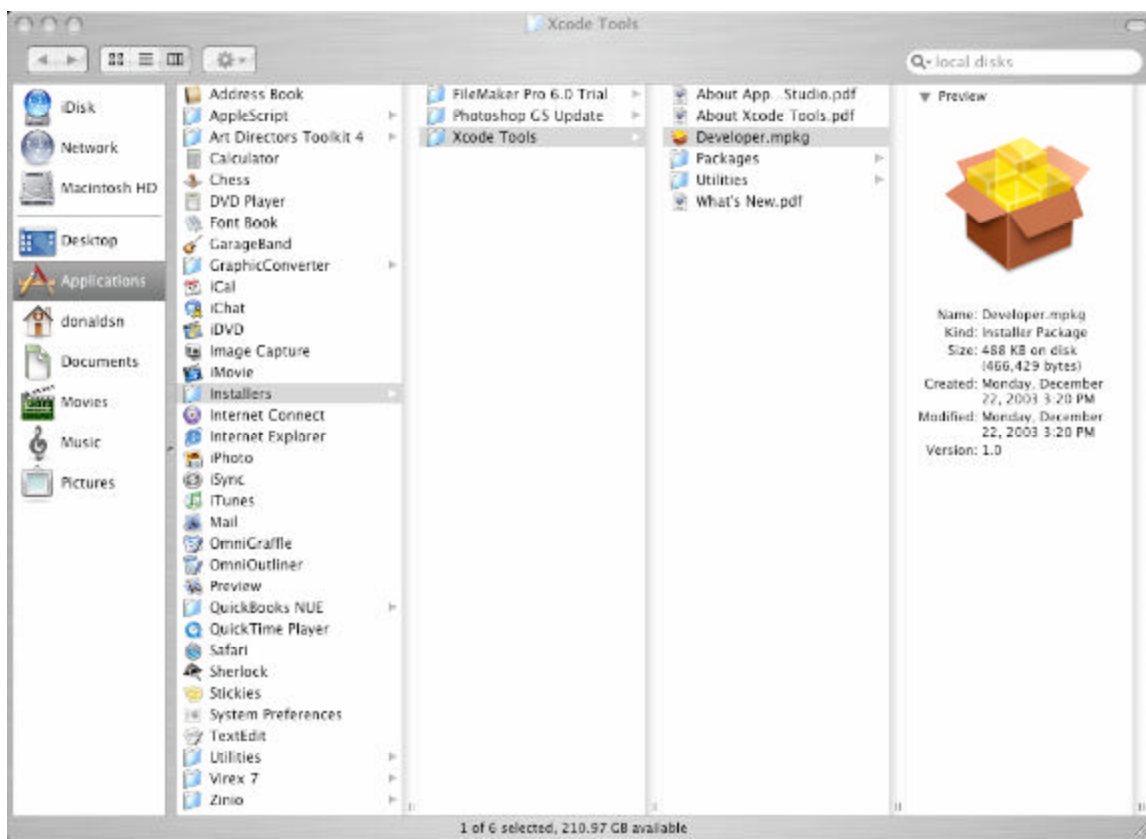
### 3.3.1 The gcc Compiler Group and Xcode

These tools are required by the Absoft version of the IBM XLF compiler. Although they come bundled with the Panther OS, they are not installed by default. Using Finder (see Figure 2), navigate to the /Applications/Installers/Xcode Tools subdirectory and double-click on the file icon named Developer.mpkg<sup>8</sup>. The Applications directory is available as an icon on the left side of the Finder window. If you can not see the Installers subdirectory after selecting the Applications menu, then change the view to either “List” or “Columns” (see Figure 3). For some reason, the Installers subdirectory is sometimes hidden in the “Icons” view.

I use the Xcode development environment at its most basic level because I have not learned how to fully exploit its capabilities with a large program like SARB. If you have time, you may want to experiment with this first. So far, I have been using Xcode to edit source code as it allows you to have several source files open at once. This is good for cutting and pasting source blocks from one subroutine to another. The Absoft version of the IBM XLF compiler distribution claims to be compatible with the Xcode software. I have not attempted to establish the SARB software as an Xcode Project yet but I imagine it is not too difficult, and it could make work with the SARB source much easier<sup>9</sup>. The source editor displays the various syntax elements in different colors, and I find this very helpful when trying to read stranger source code.

<sup>8</sup> After you double-click the Developer.mpkg file icon, follow the installation directions and take the defaults for an easy installation.

<sup>9</sup> Xcode will not be supported on the DAAC Linux cluster, so if you use it, be prepared to convert back to a make file when you port to the DAAC.



**Figure 3 Finder View Using Columns**

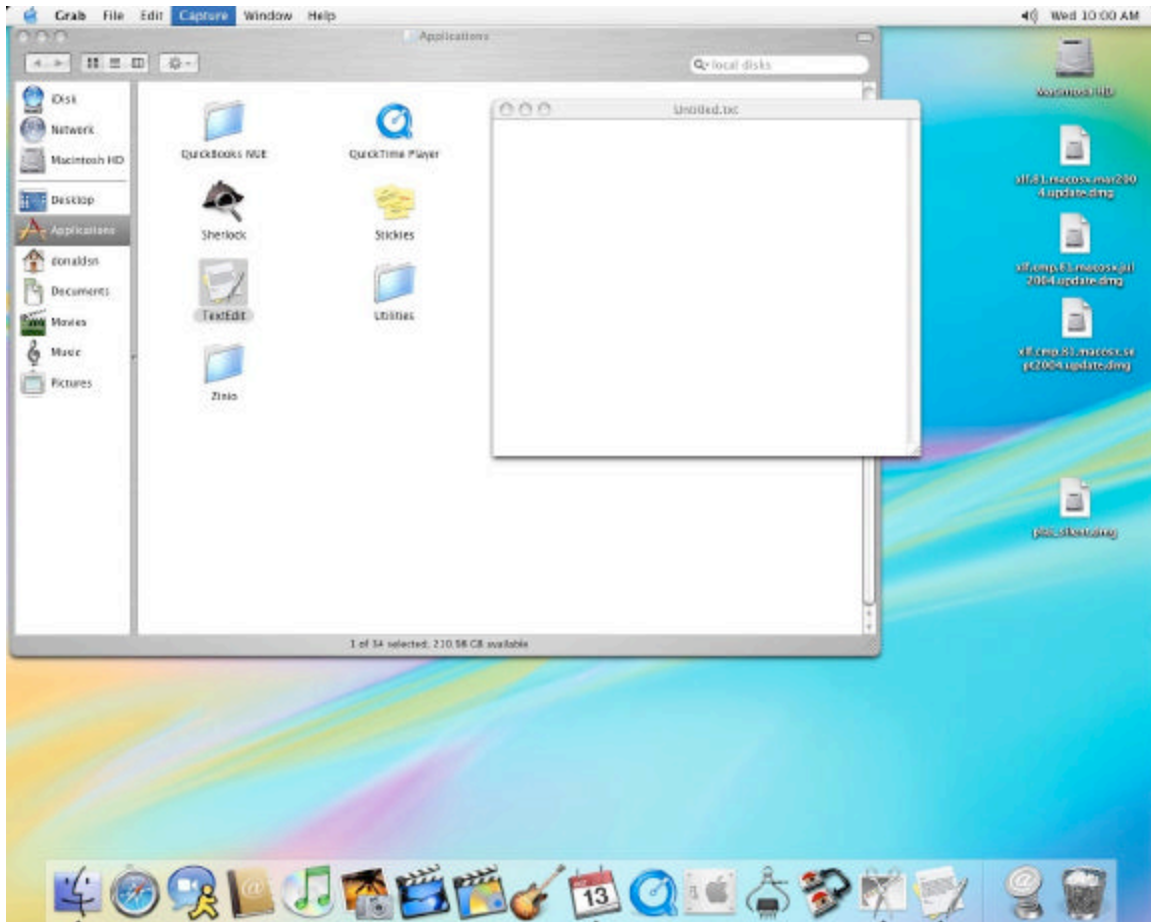
Also, there are some subtle feedback signals that take place during editing such as when you type a closing parenthesis, the matching open parenthesis is briefly flashed. Similarly, if you mistakenly type a right-parenthesis when you should be typing a left-parenthesis, the Xcode editor will beep at you. When I finish editing source code, and I am ready to build either the library or the main program, I save the modified source files and revert to the command-line make file.

If you have C source code, then you should be able to almost seamlessly switch over to the gcc compiler by updating the “CC” environment variable to gcc, and by modifying the respective make files to incorporate the appropriate gcc compiler switches. Xcode will also work with C code, and it is compatible with the gcc software.



### 3.3.2 TextEdit

In Figure 4, you can see the TextEdit icon in the Applications directory that is visible in



**Figure 4 Macintosh Display After Invoking TextEdit**

the Finder window. Double-click the TextEdit icon and the screen will look something like Figure 4. An editing window is open, ready for typing text or cut and paste operation from other applications. Also note that the TextEdit icon is now resident on the “Dock” (the 3<sup>rd</sup> from the right). The editor window can be dismissed by clicking the “X” icon in the upper left corner of the editor window. The “X” icon is the leftmost of the 3 circular buttons in the upper left corner of the window (see Figure 5).



**Figure 5 Macintosh Window Control Buttons**

If the editor window contains text that has not been saved, the left-most button control will be a black dot on a red background (see Figure 6) when you move the mouse cursor over the button. If the file is saved, the left-most button control will be a black “X” on a red background. The middle button control is a “-” character on an amber background. When you move the mouse cursor over it, the middle button allows the window to be

minimized to an icon on the Dock. The right-most button control will be a black “+” character on a green background. The right-most button control allows the window to be re-sized. I use the TextEdit window to gather text that I want to save elsewhere, or for obtaining a hard copy of text that is normally difficult, if not impossible, to print. For example, when an application program crashes, a trace-back report may get dumped to the command-line window as is the case in Figure 6.

```

Terminal — InstSARB_Drv.exe — 97x24
/CERES/sarba/rcf/pcf/sarb/CER5.1P1_PCF_Terra-FM2-MODIS_VaIR2_016020.2001100101
[cts1-97:sarba/bin/sarb] Jim% x_runsarb CER5.1P1_PCF_$INSTANCE Main > Mac21012.txt

Signal received: SIGFPE - Floating-point exception
Signal generated for floating-point exception:
  FP invalid operation

Instruction that generated the exception:
  fdivs fr05,fr05,fr08
Source Operand values:
  fr05 =  0.000000000000e+00
  fr08 =  0.000000000000e+00

Traceback:
  Offset 0x000530a8 in procedure _qftisf_
  Offset 0x0003fe58 in procedure ___fulioumulti_MOD_solver_configuration_
  Offset 0x00040560 in procedure ___fulioumulti_MOD_rad_multi_fu_
  Offset 0x00035360 in procedure ___fl_pass_interface_MOD_fl_call_
  Offset 0x00035bc8 in procedure ___fl_pass_interface_MOD_fl_pass_drv_
  Offset 0x0001183c in procedure ___fl_setup_MOD_fluxcalc_drv_
  Offset 0x0001230c in procedure ___fl_setup_MOD_inittune_drv_
  Offset 0x00006440 in procedure ___foot_drv_MOD_foot_proc_
  Offset 0x0000b198 in procedure _main
  --- End of call chain ---

```

**Figure 6 Trace-back Report in Terminal Window**

The Macintosh window manager provides the capacity to highlight text in any open window and copy it to a clipboard. The text in the terminal window can be copied and pasted to a TextEdit window, and then other information can be added from other sources. When the TextEdit file is complete, it can be saved, sent to the printer, or attached to an e-mail message.

### 3.3.3 The Activity Monitor

When running an application program that takes considerable resources, I like to use the Activity Monitor to watch the CPU utilization and the other tasks that are competing with my process. Figure 7 is a view of the Activity Manager while the SARB main process is running. There are several key statistics including the process ID, the percentage of CPU utilization, and the amount of memory that is currently needed. The other processes that are competing for resources are illustrated as well.

## CERES Conversion Guide

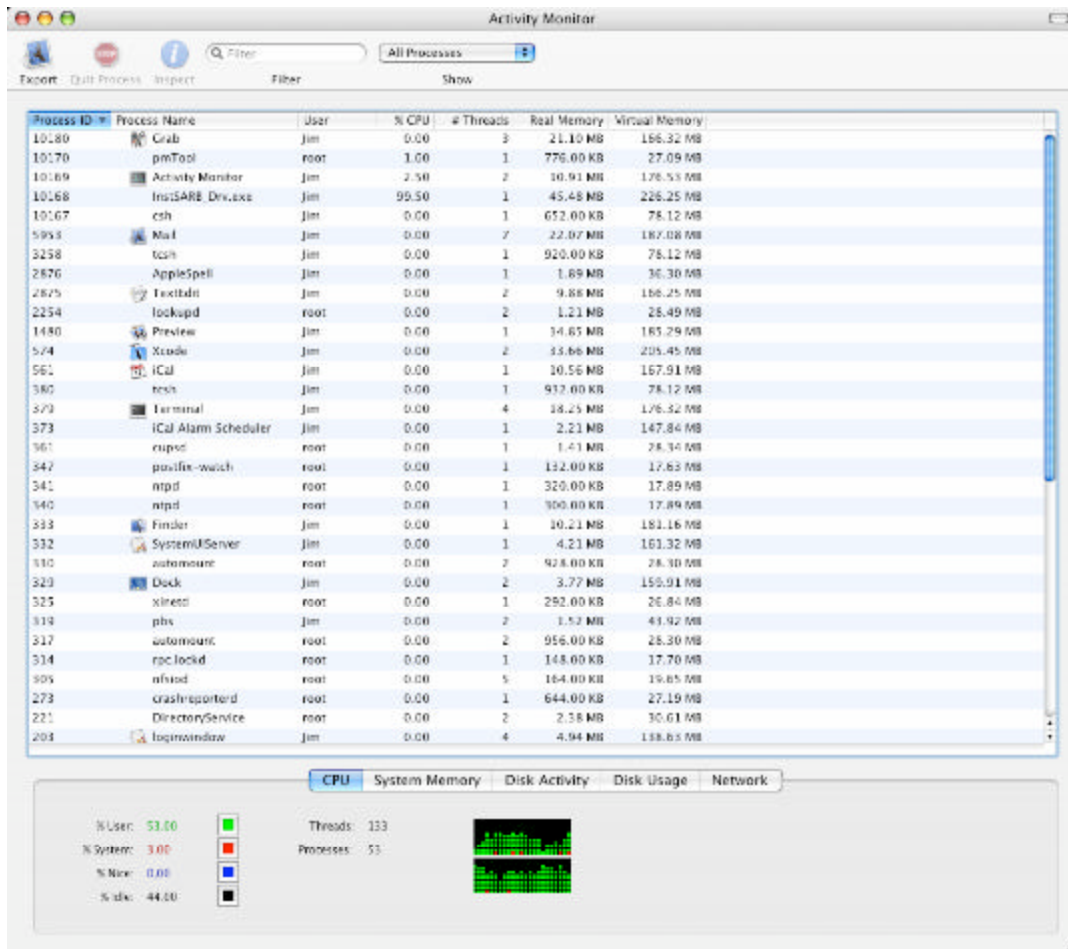


Figure 7 The Activity Manager Window

I also like to watch the dual CPU behavior using a floating window for the CPU Monitor that has “always-on-top” window status (see Figure 8).

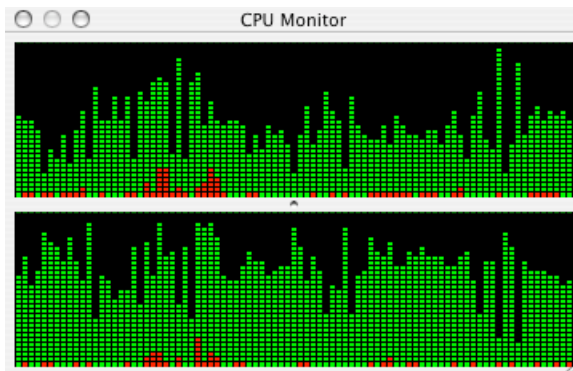


Figure 8 Floating CPU Monitor

Figure 8 illustrates the Mac G5 dual CPU's utilized for the SARB main process at about 50% each. The red blocks represent system CPU usage. A look at the Activity Monitor screen will show that InstSARB\_Drv.exe is getting approximately 100% CPU resource while at the bottom of the Activity Monitor screen you can see that the User is getting approximately 50% of the total CPU capacity (green blocks). In this example, for that moment in time, the system (red blocks) was using about 3% and the remainder was idle.

# CERES Conversion Guide



### Figure 9 InstSARB Drv.exe Open Files



### Figure 10 InstSARB\_Drv.exe Process Statistics

The Activity Monitor can be used to manage processes, and it can be used to kill a process if necessary. The Inspect icon can be activated to get statistics on Memory, Open Files, and Operating System information. Figure 9 illustrates the Open Files tab for the Inspect window.



**Figure 11 InstSARB\_Drv.exe Memory Utilization**

Figure 10 illustrates the SARB Main processor runtime statistics up to the time the process was sampled, and Figure 11 illustrates the SARB main processor memory utilization for the moment in time when the Inspect operation was performed.

### 3.4 The Console Utility

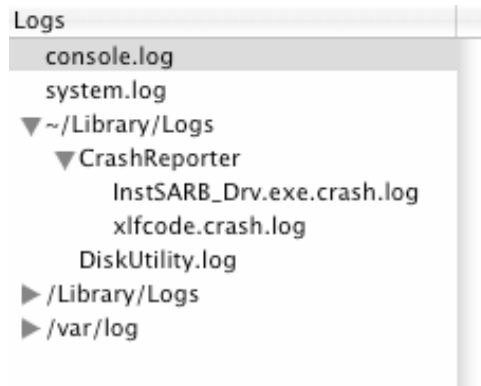
The Console Utility is worth mentioning on its own merit. Like Unix, you can open a Console window to monitor Unix-style “stderr” output, but I think the Console Utility is more valuable for the log information that it collects and retains. When you are familiarizing yourself with the Macintosh G5, take a look at the Console Utility and peruse the various logs that will already be there. The feature that I want to illustrate is the CrashReporter log.



Figure 12 Console Utility Window

To view the CrashReporter logs open the Console Utility by selecting the Applications folder on the left side of the Finder window (see Figure 2). Then, double-click on the Utilities icon to view most of the utilities that are available on the G5. One of these utilities will be the Console Utility. Figure 12 illustrates the Console Utility window as it might look the first time you open it. In Figure 12 the Console log is selected and there is a history of entries that have been accumulating. To view the CrashReporter log, click on the ~/Library/Logs, dark triangle in the upper left corner of the window. Figure 13 shows

the log selector controls expanded after a single click on the ~/Library/Logs entry. In this example, the CrashReporter entry has already been selected, and we can see that there are two entries under the CrashReporter. The entry that I want to discuss is the InstSARB\_Drv.exe.crash.log log. A single click on the InstSARB\_Drv.exe.crash.log entry will select the corresponding log file, and the log text will be displayed in the pane on the right side of the window.



**Figure 13 Log Selection Controls**

Figure 14 illustrates the latest crash information for the SARB main processor. This particular crash was the result of an exception generated by an invalid floating point operation, or more precisely, a divide by zero.

Every time your application crashes, a log entry will document the crash in the CrashReporter log section of the Console Utility. This behavior happens silently, and if you never check for a crash log, the only information available is the output in the terminal window in which you started the application<sup>10</sup>. A quick look back at Figure 6<sup>11</sup> will reveal that the trace-back information provided in the terminal window output is lacking the hexadecimal offset data that is necessary for pinpointing the exact line of source code that was responsible for the crash. In Figure 14 we can see that this precious information is present at the end of each entry in the trace-back list<sup>12</sup>. In this example, we can see that procedure (subroutine) “add” caused the crash, and the offending instruction resides at hexadecimal offset 0x2E0. This means that if we have an object listing<sup>13</sup> for the source file that owns subroutine add, we can determine what source line number in subroutine add is causing the problem. The trace-back information is very valuable. For example, sometimes it is very important to follow the call chain back to one of the callers that might be passing a bad parameter along the chain such that the bad parameter does not cause any problems until the last procedure in the chain tries to use the parameter.

<sup>10</sup> I am excluding whatever diagnostic information you have chosen to output on your own.

<sup>11</sup> I apologize that Figure 6 is not the same crash as the one logged in Figure 14.

<sup>12</sup> The trace-back list is perhaps better described as the call stack.

<sup>13</sup> An Object Listing is the result of implementing the -qlist compiler switch during the compilation of the source file in question.



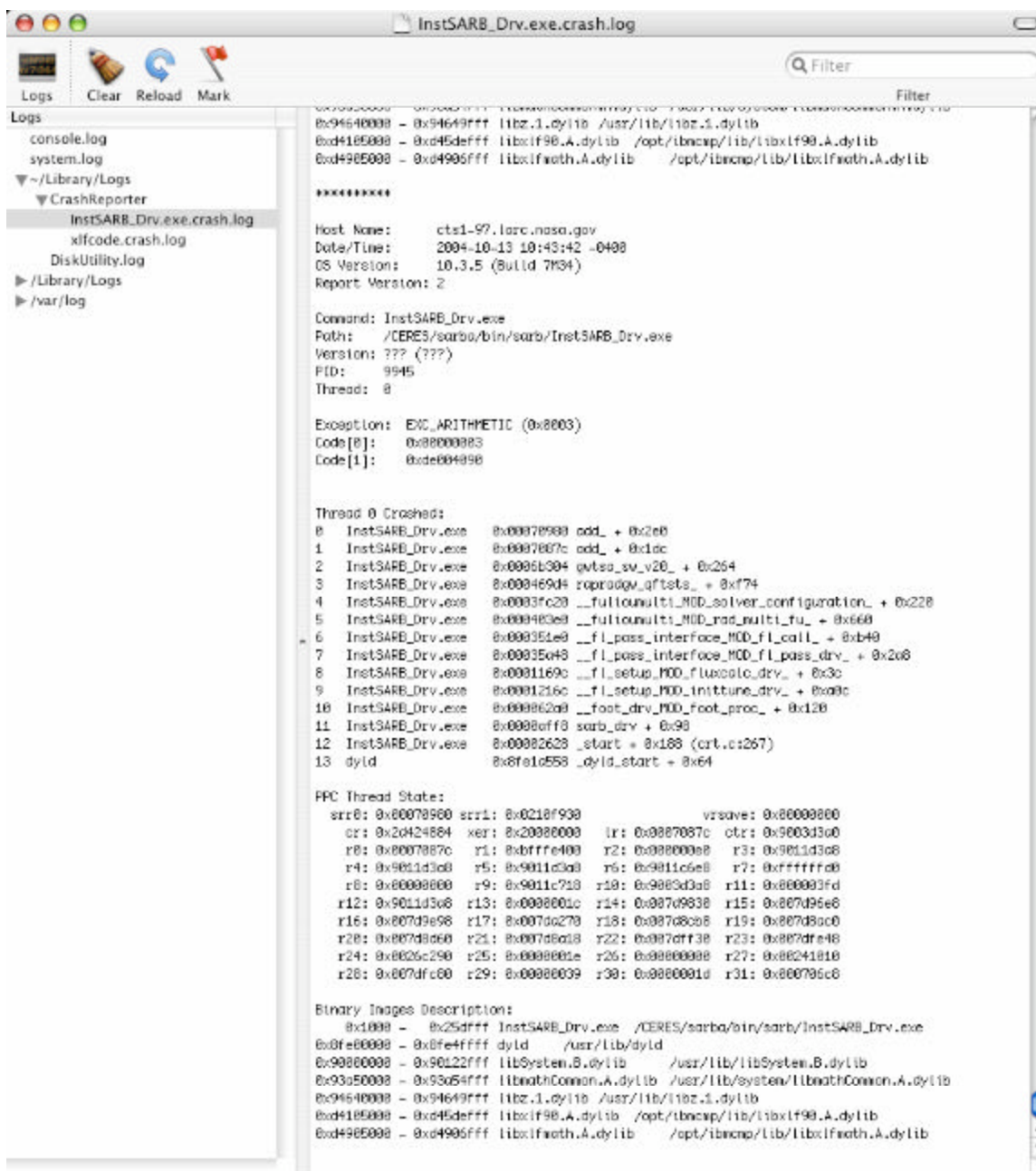


Figure 14 InstSARB\_Drv.exe.crash.log Window

In case this information is new to you, I will spend a little more effort identifying some of the components of the crash log file.

- The start of the log is indicated by the string of asterisks
- The first section of the log identifies the Host Name, Date and Time of the crash, and the Operating System Version and build number
- The second section of the log identifies the executable program, where it lives, its version, Process ID, and the number of the thread that crashed
- The third section describes the exception that occurred
- The forth section is the trace-back list or call stack
- The fifth section is the register state of the Power PC at the time of the crash



- The sixth section describes the binary images that make up the executable and where the images live

Virtually all of this information is very helpful for finding the cause of the crash, and in a later section I will walk through an example of locating the source line given a crash log.

### **3.5 The IBM XL FORTRAN Compiler (IBM XLF)**

I do not mean to present a tutorial on the IBM XLF compiler here. You need to experiment with the compiler and read the User's Guide and Language Reference manuals that are distributed with the compiler. I am going to present the compiler switch configurations that I used to port the SARB main processor to the Mac G5 platform. There are a few different compiler switch configurations to build the source files depending on what you are trying to accomplish. I take a phased approach to converting a CERES subsystem. The phases are:

1. Initial build and language level assessment
2. Complete build and sweep for array bounds violations
3. Complete build and sweep for invalid floating point operations
4. Complete build with safe optimization and benchmark comparison

Before starting with the various phases of building and checking out the subsystem code, an analysis of the source code should be conducted to assess what FORTRAN standard was applied during code development. For SARB it appears that the code is a graduation from FORTRAN 77 to FORTRAN 90. Version 8.1 of the IBM XLF compiler supports the FORTRAN 77, FORTRAN 90, and FORTRAN 95 standards.

#### **3.5.1 What Standard Should I Use?**

For the SARB conversion, I experimented with FORTRAN 95 and FORTRAN 90. I can not honestly say that I noticed a difference but when I had hardware problems with the Mac G5, I reverted to using the FORTRAN 90 standard for the rest of the SARB conversion process. If you would rather use the FORTRAN 95 standard from the start, it should be fine.

SARB and CERESlib both use a C-shell script called `ceres-env.csh` to identify environment variables that support compiling, linking and execution of science code. The script defines an environment variable named "F90". I set F90 to the value, `$XLF` which is an indirection since it refers to yet another environment variable, `XLF`. I set `XLF` to the value, `"/opt/ibmcmp/xlf/8.1/bin/xlf90"`. This path specification should work for Darwin and for Linux unless the Linux system administrator elects not to install the IBM XLF compiler at the preferred, default location<sup>14</sup>.

#### **3.5.2 Initial Build and Language Level Assessment**

The initial build on the target platform can be frustrating depending on the initial compiler settings. I prefer not to flood the compilation results with lots of warning messages that are concerned with the usage of tab characters and other obvious extensions of the FORTRAN 90 standard. Consequently, I choose to employ compiler settings that are typical to those that I might choose for routine development. This will

---

<sup>14</sup> This very well may be true for a cluster setup.

flush out the obvious differences between the source platform compiler and the IBM XLF compiler.

In section 3.5.1 What Standard Should I Use?, I mentioned that SARB and CERESlib both use a C-shell script called `ceres-env.csh`. The `ceres-env.csh`<sup>15</sup> sets environment variables that define the IBM XLF compiler environment. Table 1 identifies the parameters that we will need set prior to invoking a typical CERES make file.

<i>Environment Variable</i>	<i>Value</i>	<i>Description</i>
<b>XLF</b>	<code>/opt/ibmcmp/xlf/8.1/bin/xlf90</code>	Path to the IBM XLF compiler executable
<b>F90</b>	<code>\$XLF</code>	Identifies the FORTRAN 90 compiler that is currently in use
<b>F90COMP</b>	<code>- qxlf90=nosignedzero,autodealloc -O2 -c -qextname -qsuffix=f=f90 -qmaxmem=32768</code>	Typical FORTRAN 90 switch settings for optimized development work
<b>FCOMP</b>	<code>- qxlf90=nosignedzero,autodealloc -O2 -c -qextname -qfixed=80 -qmaxmem=32768</code>	Typical FORTRAN 90 switch settings for compiling FORTRAN 77 code
<b>F90LIBDIR</b>	<code>/opt/ibmcmp/xlf/8.1/lib</code>	Path to IBM XLF libraries
<b>F90LIB</b> <b>F90LOAD</b> <sup>16</sup>	<code>\$F90LIBDIR/libxlf90.dylib</code>	XLF F90 library Flags passed to the Darwin loader by IBM XLF

**Table 1 - IBM XLF Environment Variable Settings**

For now we will focus on F90COMP and FCOMP, and you can see that the compiler switches overlap quite a bit with the exception of the “suffix” and the “fixed” switches. Let’s take a look at each switch setting:

- **-qxlf90=nosignedzero,autodealloc** provides compatibility with the FORTRAN 90 standard. For the xlf90 setting the defaults are **nosignedzero** and **noautodealloc**. **nosignedzero** prevents formatted output from specifying a minus sign when the result is essentially zero. It also effects how the **SIGN(A,B)** function handles signed real 0.0. **autodealloc** tells the compiler to automatically de-allocate objects that are declared locally without either the **SAVE** or the **STATIC** attribute, and that have a status of currently allocated when the subprogram terminates.
- **-O2** performs a set of optimizations that are intended to offer improved performance without an unreasonable increase in time or storage that is required for compilation.
- **-c** tells the compiler to produce an object file instead of an executable file.
- **-qextname** adds an underscore to the names of all global entities. The IBM XLF compiler does not do this by default, and it is necessary for compatibility with the PGS Toolkit and for calling C procedures from FORTRAN.

<sup>15</sup> `ceres-env.csh` also defines the other libraries and directories required by the application but here we are focusing on the IBM XLF compiler.

<sup>16</sup> I perform conversion work with F90LOAD defined but blank to avoid getting the source symbols stripped out of the executable file. For production we will add in “-static -s” to guarantee that we get a static link and that the source symbols are stripped.

- **-qmaxmem=32768** (Kbytes) limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations. A value of -1 allows optimization to take as much memory as it needs without checking for limits. The default value with O2 set is 2048 Kbytes. With the default setting, the compiler frequently emits a warning message saying that more optimization is possible for the compiled object. 32768 Kbytes seems to eliminate most, if not all, of the warnings.
- **-qsuffix** specifies the source-file suffix. For the source file suffix the IBM XLF compiler defaults to “.f”. The **-qsuffix=f=f90** setting tells the compiler to look for source files with the suffix, “.f90”. If you have mixed F90 and F77 source files (.f90 and .f, respectively), then you must design your make files with rules such that the F90COMP settings are used for the .f90 files and the FCOMP settings are used for the .f files. Note that the FCOMP settings do not incorporate the suffix switch since XLF defaults to the .f syntax.
- **-qfixed=80** indicates that the input source code is in fixed source form. The xlf90 compiler defaults to free form source code, so we need to tell it otherwise when compiling FORTRAN 77 code. The **-qfixed** switch will default to 72 columns, so I have increased the column width to 80 to avoid compiler errors in the F77 source code.

We have defined the explicit compiler switches, but what are the switch settings that we did not override? Good question. Here is an earnest attempt to nail that down the compiler switches that the XLF compiler lists as active given our explicit specifications:

- **-qcr** allows you to control how the compiler interprets the Carriage Return character. This allows compilation of code written using a Mac OS or DOS/Windows editor.
- **-qescape** specifies how the backslash is treated in character strings, Hollerith constants, H edit descriptors, and character string edit descriptors. The default setting (-qescape) is to treat the backslash as an escape character.
- **I4 TBD**
- **-qnolist** specifies not to produce the object section of the source listing. The object section is critical for decoding trace-back information.
- **-qOBJect** to produce an object file as opposed to stopping immediately after checking the syntax of the source files.
- **-qnosource** specifies to not produce the source section of the listing.
- **SWAPOMP TBD**
- **-qunwind** specifies that the compiler will preserve the default behavior for saves and restores to volatile registers during a procedure call.
- **-qzerosize** improves the performance of F77 and some FORTRAN 90 programs by preventing checking for zero-sized character strings and arrays.
- **-qspillsize=512** specifies the size of internal program storage areas. It defines the number of bytes of stack space to reserve in each subprogram, in case there are too many variables to hold in registers and the program needs temporary storage for register contents. If this option is needed, a compiler message will be issued.
- **-qalias=aryovrlp:pteovrlp:std:nointptr** indicates categories of aliasing. **aryovrlp** indicates compilation units may contain array assignments between storage-associated arrays. **pteovrlp** indicates pointee variables may be used to refer to any data objects that are not pointee variables, or that two pointee

variables may be used to refer to the same storage location. **std** indicates the compilation units contain no nonstandard aliasing. **nointptr** indicates that compilation units do not contain any integer POINTER statements.

- **-qalign=no4k:struct=natural** specifies the alignment of data objects in storage. no4k specifies not to align large data objects on page (4 KB) boundaries. struct=natural specifies that objects of derived types are stored with sufficient padding that components will be stored on their natural alignment boundaries, unless storage association requires otherwise.
- **-qarch=ppcv** generates instructions for generic PowerPC chips with AltiVec vector processors.
- **-qautodbl=none** does not promote or pad any objects that share storage.
- **-qdirective** turns on the default trigger constant IBM\*.
- **-qflag-i:i** specifies the listing severity and terminal severity for diagnostic messages, respectively. Informational Messages (i) are the lowest level, so **-qflag:i** guarantees that no messages are missed.
- **-qfloat=nocomplexgcc:nofltint:fold:maf:nonans:norrm:norsqrt:nostrictnmaf** selects different strategies for speeding up or improving the accuracy of floating-point calculations. The sub-options have the following meanings:
  - **nocomplexgcc** uses Mac OS X conventions when passing or returning complex numbers.
  - **nofltint** does not allow the use of an inline sequence of code in lieu of a call to a library function for floating-point to integer conversions.
  - **Fold** causes the compiler to evaluate constant floating-point expressions at compile time, which may yield slightly different results from evaluating them at run time.
  - **maf** makes floating-point calculations faster and more accurate by using multiply-add instructions where appropriate.
  - **nonans** prevents the usage of **-qflttrap=invalid:enable** to detect exception conditions that involve signaling NaN values.
  - **norrm** prevents the turn OFF of compiler options that require the rounding mode to be the default, round-to-nearest, at run time.
  - **norsqrt** prevents the replacement of division by the result of a square root with multiplication by the reciprocal of the square root.
  - **nostrictnmaf** prevents the turn OFF of floating-point transformations that are used to introduce negative MAF instructions, as these instructions do not preserve the sign of a zero value.
- **-qfree** specifies FORTRAN 90 free source form<sup>17</sup>.
- **-qhalt=s** prevents the compiler from generating an object module when compilation fails.
- **-qieee=Near** tells the compiler to round to the nearest representable number when it evaluates constant floating-point expressions at compile time.
- **-qintsize=4** specifies the default INTEGER and LOGICAL data entities for which no length or kind is specified.
- **-qlanglv=extended** causes the compiler to accept the full FORTRAN 95 language standard plus all extensions, effectively turning off language-level checking.

<sup>17</sup> This is the case for a .f90 source file.

- **-qpic** causes the compiler to generate Position Independent Code (PIC) that can be used in shared libraries.
- **-qrealsize=4** sets the default size of REAL (4 bytes), DOUBLE PRECISION (8 bytes), COMPLEX (4 bytes), and DOUBLE COMPLEX (8 bytes) values that are declared without a length or kind.
- **-qtune=g5** generates object code that is optimized for G5 processors. This is currently equivalent to specifying **-qtune=ppc970**.
- **-qunroll=auto** instructs the compiler to perform basic loop unrolling.
- **-qxflag** causes the compiler to default to allowing 66 significant characters on a source line after column 6. A tab in columns 1 through 5 is interpreted as the appropriate number of blanks to move the column counter past column 6.
- **-qxlf77=blankpad:nogedit77:nointarg:nointxor:leadzero:nooldboz:nopersistant:nosofteof** provides compatibility with XL FORTRAN for AIX Versions 1 and 2. The sub-options are explained briefly below:
  - **blankpad** specifies padding for allowed for internal, direct access, and stream-access files.
  - **nogedit77** prevents the usage of F77 semantics for the output of REAL objects with the G edit descriptor.
  - **nointarg** prevents the conversion of integer arguments of an intrinsic procedure to the kind of the longest argument if they are of different kinds.
  - **nointxor** prevents treating .XOR. as a logical binary intrinsic operator.
  - **leadzero** produces a leading zero in real output under the D, E, L, F, and Q edit descriptors.
  - **nooldboz** prevents turning blanks into zeroes for data read by B, O, and Z edit descriptors, regardless of the BLANK= specifier or any BN or BZ control edit descriptors.
  - **nopersistant** prevents the saving of addresses of arguments to subprograms with ENTRY statements in static storage.
  - **nosofteof** prevents READ and WRITE operations when a unit is positioned after its endfile record.

There are many other compiler switches that are not mentioned in this context. For the most part, they are either not in play, implied or secondary to a setting above, or they are not appropriate for the context established by the explicitly defined switches selected in Table 1 via FCOMP and F90COMP.

For the initial build we would like to ferret out and correct any severe compiler errors and the significant warnings; that is, the warnings that indicate the compiler has not failed but conversely, the code will probably not produce the correct result either. Once the initial build phase is completed, we can consistently compile the source with no errors and a minimum of warning messages. We are then in a position to assess the language level of our application software.

To assess the language level of our application software we can add **-qlanglvl=suboption** to our F90COMP and FCOMP environment variable definitions. The suboption choices are illustrated in Table 2.

<i>Suboption</i>	<i>Description</i>
<i>77std</i>	Accepts the language that the ANSI FORTRAN 77 standard specifies and reports anything else as an error.
<i>90std</i>	Accepts the language that the ISO Fortran 90 standard specifies and reports anything else as an error.
<i>90pure</i>	The same as <b>90std</b> except that it also reports errors for any obsolescent Fortran 90 features used.
<i>95std</i>	Accepts the language that the ISO Fortran 95 standard specifies and reports anything else as an error.
<i>95pure</i>	The same as <b>95std</b> except that it also reports errors for any obsolescent Fortran 95 features used.
<i>extended</i>	Accepts the full Fortran 95 language standard plus all extensions, effectively turning off language-level checking.

Table 2 - IBM XL Fortran Compiler Language Level Suboptions

A good first choice would be to add **-qlanglvl=95std** to the F90COMP and FCOMP definitions and re-build the program. For the SARB conversion I found that the majority of the language level warning messages were related to the usage of the tab character. If there are errors, then they should be corrected to conform to the Fortran 95 standard because there is a fair chance that the same source code will be problematic in the conversion effort. When the language level assessment is complete, modify the F90COMP and FCOMP definitions to remove the **-qlanglvl=suboption** switch in preparation for the next phase.

### 3.5.3 Complete Build and Sweep for Array Bounds Violations

I learned the hard way not to assume that the code developer produced code that has no illegal array references. Consequently, my first test case is not to generate results for comparison, but rather to ferret out the compiler bounds violations; at least the ones that will occur with the selected test case. I completely re-build the source code, adding the following compiler switches to the ones described in section 3.5.2 Initial Build and Language Level Assessment:

- **-C** checks at compile time for any array references that are out of bounds and increases the severity level to SEVERE for such occurrences. At run time, if an array reference goes out of bounds, the program generates a **SIGTRAP** signal. By default, this signal ends the program and produces a core dump if the main program has been compiled with the **-qsigtrap** switch specified.
- **-qsigtrap** causes the main program, when compiled, to include a trap handler to catch **SIGTRAP** and **SIGFPE** signals without having to call the **SIGNAL** subprogram. The default trap handler is **xl\_\_trce** but you can supply your own trap handler by specifying **-qsigtrap=trap\_handler**.
- **-qlist** causes the compiler to generate the object portion of a source listing. The object portion is critical for using the trace-back data to determine what line of source code caused an exception to occur.
- **-qsource** causes the compiler to generate a source listing file (with line numbers) with the same name as the source file but using the file suffix, .lst.

The **-C** and **-qsigtrap** switches facilitate stopping the program on an array bounds violation with the added benefit of a trace-back dump in the Console CrashReporter log for the application in question. The **-qlist** and the **-qsource** switches provide the reference information we will need to determine the exact line of source code that causes the array bounds violation.

Each time an array bounds violation occurs, correct the problem either by inventing a workaround fix or by implementing a permanent correction for the problem. Make this same modification to the benchmark code on the source platform in preparation for the comparison activities. Execute the test case on the target platform until the test case runs to a normal completion. With array bounds checking turned on, the test cases will take significantly longer to execute.

### 3.5.4 Complete Build and Sweep for Invalid Floating-Point Operations

I learned the hard way not to assume that the code developer produced code that has no invalid floating-point operations. Consequently, my second test case is not necessarily to generate results for comparison, but rather to ferret out the invalid floating-point operations; at least the ones that will occur with the selected test case. I completely re-build the source code, adding the following compiler switches to the ones described in section 3.5.2 Initial Build and Language Level Assessment:

- **-qinitauto=7fbffff** causes AUTOMATIC variables to be initialized with the given value. When the value 0x7FBFFFFF is stored in a REAL(4) variable, we have a single precision signaling NaN. **-qinitauto=7ff7ffff** will cause REAL(8) AUTOMATIC variables to store double precision signaling NaN values.
- **-qfloat=nans** allows you to use the **-qflttrap=invalid:enable** option to detect exception conditions that involve signaling NaN values.
- **-qflttrap=zero:inv:en** defines the types of floating-point exception conditions to detect at run time. The program receives a **SIGFPE** signal when the corresponding exception occurs. I specify the following sub-options:
  - **zero** is short for **ZERODivide** and facilitates the detection of floating-point division by zero *if exception checking is enabled*.
  - **inv** is short for **INValid** and facilitates the detection of floating-point invalid operations *if exception checking is enabled*.
  - **en** is short for **ENable** and causes checking for the specified exceptions to be turned on. This sub-option must be included to turn on exception trapping without modifying the source code.
- **-qsigtrap** causes the main program, when compiled, to include a trap handler to catch **SIGTRAP** and **SIGFPE** signals without having to call the **SIGNAL** subprogram. The default trap handler is **xl\_\_trce** but you can supply your own trap handler by specifying **-qsigtrap=trap\_handler**.
- **-qlist** causes the compiler to generate the object portion of a source listing. The object portion is critical for using the trace-back data to determine what line of source code caused an exception to occur.
- **-qsource** causes the compiler to generate a source listing file (with line numbers) with the same name as the source file but using the file suffix, **.lst**.

The **-qfloat=nans**, **-qflttrap=zero:inv:en**, and **-qsigtrap** switches facilitate stopping the program on an invalid floating-point operation with the added benefit of a trace-back dump in the Console CrashReporter log for the application in question. The **-qlist** and the **-qsource** switches provide the reference information we will need to determine the exact line of source code that causes the invalid floating-point operation.

Each time an invalid floating-point operation occurs, correct the problem either by inventing a workaround fix or by implementing a permanent correction. Make this same modification to the benchmark code on the source platform in preparation for the comparison activities. Execute the test case on the target platform until the test case runs to a normal completion. With invalid floating-point operation trapping enabled, the test cases will take significantly longer to execute.

### 3.5.5 Complete Build for Safe Optimization and Results Comparison

Once we have eliminated array bounds violations and invalid floating-point operations, the application software should be ready for a comparison run. Modify the F90COMP and the FCOMP<sup>18</sup> environment variables to conform to the settings identified in Table 1 and re-build the application software. Add any other compiler switches that you think will be appropriate. If you are in doubt about this, I suggest keeping it simple until you have seen the comparison results. Then you may have a better idea of what settings may need tweaking. Also, check the compiler settings on the source platform to make sure there are no special settings<sup>19</sup> required. If there are special settings, then use the IBM XL Fortran User's Guide to determine what switch settings are required to arrive at an equivalent setup.

Build the program on the source platform and run the test case to establish the benchmark files to be used for comparison. Once the test case has successfully executed on the source platform then build the program on the target platform and run the same test case to generate the comparison files<sup>20</sup>.

---

<sup>18</sup> Or your equivalent means to set the compiler switches.

<sup>19</sup> If there are extreme optimization switches on the source platform, leave them in place but do not try to duplicate them on the target for your first comparison run. Once you get a good comparison, then you can experiment with higher levels of optimization.

<sup>20</sup> I have not included the compiler switches for high levels of optimization. The XLF compiler has a good complement of switches for advanced optimization. Once the conversion is successful, it is a good idea to experiment with higher levels of optimization but that is not my goal here. The *XL Fortran Advanced Edition for Mac OS X User's Guide, Version 8.1* has an excellent approach to optimization in the section, "Optimizing XL Fortran Programs".



### 3.6 The Darwin Static Linker (ld) and Archive Library (ar) Tools

On the Mac, under Darwin, the static linker (/usr/bin/ld) is used by the compiler driver (gcc, XLF, E.T.C.) and as a standalone tool to combine Mach-O executable files. The static linker can be used to bind programs either statically or dynamically. The archive library tool, ar (/usr/bin/ar), is used to create and maintain static archives that are suitable for use with ld. If the -s switch is not used when ar creates or updates a static library, then the ranlib (/usr/bin/ranlib) tool can be employed to write an object-file index into the archive. As an alternative, running “ar s” is equivalent to running ranlib on the archive. Also, incorporating the s<sup>21</sup> switch with the r switch when ar creates a static library will include the object-file index in the archive library without requiring a second step.

If you want to learn more about the Mach-O runtime architecture, then I suggest acquiring and reading *Mach-O Runtime Architecture for Mac OS X version 10.3* by Apple Computer, Incorporated. This document is very useful for application developers.

### 3.7 The Make Utility

Darwin incorporates the GNU make utility to maintain groups of programs. During the SARB conversion I found the SGI make files to be completely compatible with the GNU make utility<sup>22</sup>. The make file for the SARB library (SARBlib) uses the ar utility to create SARBlib, and I did have to modify the arguments to ar to avoid an error. But this was due to a difference in the ar utilities between SGI and the Mac G5, and not in the make utility.

## 4 A Process for Porting CERES Science Code

The first step in any process is to define the process! Formulate a detailed plan for how you are going to accomplish your goal<sup>23</sup>. Formulating a detailed plan will drive you to consider what tools and other resources you will need to get the job done. Once you have a draft plan and a preliminary estimate of the required resources, you are ready to start crafting a schedule. If you have more than one person available to support the conversion, then some tasks can be performed in parallel. Developing the schedule may cause you to refine the plan and the resource estimate. Plan, resource, and schedule determination is an iterative process in itself. Finally, when you have developed a strong plan, you will know what the required resources are, and you will have a reasonable idea of how long it will take.

### 4.1 Plan Your Work

If you were porting a simple program with three or four subroutines, then you could consider simply moving your source code to the target system with no forethought or formal plan. The task would be straight forward; compile the source code, link, and execute a test case.

---

<sup>21</sup> This is different syntax than what is done on the SGI platform.

<sup>22</sup> To convert SARB, I needed to install the PGS Toolkit and CERESlib, both of which depend heavily on make files.

<sup>23</sup> We already know what the requirements are; developing a detailed plan incorporates the concepts of requirements analysis and specification.

## CERES Conversion Guide

This guide is directed at CERES science code developers who are converting their subsystem software to run on Power PC 970 platforms like the Macintosh G5 and the Linux-based Power PC 970 cluster at the Langley DAAC. For this level of effort, a Conversion Plan should be prepared such that the conversion process is well understood before the effort begins.

Your conversion plan should identify and describe all the subsystem components<sup>24</sup> that are to be converted to run on the target platform. The component descriptions should include enough detail to enable a software engineer, who is unfamiliar with the subsystem, to follow the plan to implement the conversion process. Part of the identification process is a detailed description of the subsystem dependencies. For example, if I asked you, “what would be the first component of your subsystem to get converted to the target platform?”, you would probably have a pretty good idea what that component is. In the case of SARB, a reasonable answer is, “the library, of course”. For SARB, the four SARB Product Generation Executives (PGEs) all require the SARB library, SARBlib, to be in place before they can be linked in preparation for execution.

Continuing the example one more step; my next question would be, “what has to be in place on the target platform for your first component to be built?”<sup>25</sup> In the case of SARBlib, one answer is, “the PGS Toolkit and the CERES library”. I suspect that this is the case for every CERES subsystem. But, let’s assume that the Toolkit and CERESlib are already converted and installed on the target. Carrying our example one more step, I would ask, “Does the component in question require a make file or some other kind of script to build it?” If a component does require a make file or script for the build process, will the make file or script run on the target platform? Fortunately, only the most obscure script language will require serious levels of effort in the conversion process from the SGI platform to either Unix-based or Linux-based PowerPC platforms.

I’m saying that you need to write a thorough conversion plan. I took about 6 weeks to write the SARB Conversion Plan after I spent about 4 weeks familiarizing with the SARB SGI implementation. If you are the lead for your subsystem, then you will have an advantage as you should have a thorough knowledge of your subsystem and access to documents that will be useful for writing the conversion plan. I am including an outline here mostly because I really agonized over the format of the SARB Conversion Plan. Now, in hindsight, I can see that the SARB Conversion Plan is a very good starting point for a subsystem reference manual. How good is your existing subsystem reference manual? Is it up to date? Can a qualified software engineer take your subsystem reference manual and understand the subsystem components?<sup>26</sup> Fortunately, these are rhetorical questions! As an example, I have included the Table of Contents from the SARB Conversion Plan in Listing 1:

Table of Contents
Table of Tables
Table of Figures
Acronyms and Abbreviations
1.0 Introduction

<sup>24</sup> A subsystem’s major components are its dedicated library(s) and each of its Product Generation Executables.

<sup>25</sup> If it’s a library like SARBlib we need to run the ar utility on the compiled library modules. If it’s an executable, we need to link it using ld.

<sup>26</sup> Could I have used your subsystem reference manual to create a conversion plan?

## CERES Conversion Guide

- 2.0 Objective
- 3.0 SARB: The Subject of the Conversion
  - 3.1 Subsystem Dependencies
    - 3.1.1 CERES Library
    - 3.1.2 SDP Toolkit
    - 3.1.3 SARB Library
  - 3.2 The SARB Subsystem Components
    - 3.2.1 PGE CER5.0P1 – Subsystem 5.0 Monthly Preprocessor
      - 3.2.1.1 The Surface Albedo Monthly Preprocessor Input Files
      - 3.2.1.2 The Daily MODIS Aerosol Interpolation Monthly Preprocessor Input Files
      - 3.2.1.3 The Preprocessor Output Files
    - 3.2.2 PGE CER5.1P1 – Subsystem 5.0 Main Processor
      - 3.2.2.1 CER\_SSFB – Hourly Binary Single Satellite Footprint (SSFB)
      - 3.2.2.2 CER\_MOA – Hourly Meteorological, Ozone, and Aerosol Ancillary Data Set
      - 3.2.2.3 CER\_HMPSAL – Monthly Surface Albedo History File
      - 3.2.2.4 CER\_HMAER – Interpolated Daily MODIS Aerosol Files
      - 3.2.2.5 CER\_SSFA – Hourly Binary SSF Supplemental Aerosol Files
      - 3.2.2.6 MATCH\_TERRA\_AOTS\_MODIS – Daily MATCH Climatological Aerosol
      - 3.2.2.7 The Main Processor Output Files
    - 3.2.3 PGE CER5.3P1 – Subsystem 5 HDF Post-processor
    - 3.2.4 PGE CER5.4P1 – Subsystem 5 Monthly QC Processor
      - 3.2.4.1 CER\_CRS and CER\_CRSEB Input Pairs
      - 3.2.4.2 CER\_HQCR – Hourly QC Report Files
      - 3.2.4.3 CER\_HMAVAIL – Main Processor QC Report Summary
  - 3.3 The SARB Subdirectory Hierarchy
    - 3.3.1 The SARB Source Files
    - 3.3.2 The SARBLib Source and Archive Subdirectory
    - 3.3.3 The SARB Status Message Files (SMF) Subdirectory
    - 3.3.4 The SARB Resource Control File (RCF) Subdirectory
    - 3.3.5 The SARB Executables Subdirectory
    - 3.3.6 The SARB Data Subdirectory
      - 3.3.6.1 The ../data/input/sarb Subdirectory
      - 3.3.6.2 The ../data/ancillary Subdirectory
      - 3.3.6.3 The ../data/scr Subdirectory
      - 3.3.6.4 The ../data/out\_comp Subdirectory
      - 3.3.6.5 The ../data/errlogs Subdirectory
      - 3.3.6.6 The ../data/out\_exp Subdirectory
      - 3.3.6.7 The ../data/runlogs Subdirectory
  - 3.4 Non-SARB Subdirectory Data Dependencies
    - 3.4.1 Input Files From Inversion
    - 3.4.2 Input Files From Clouds
- 4.0 PPC970: The Target Platform
  - 4.1 The BladeCenter Node
  - 4.2 The Mac G5 Desktop
- 5.0 Converting SARB: The Process
  - 5.1 The Library Installation Phase
    - 5.1.1 Install the SDP Toolkit on the Target Platform
      - 5.1.1.1 Repeat Recent Attempt to Port Mac Darwin Configuration
      - 5.1.1.2 Attempt to Convert the SGI or SUN Toolkit Configuration
      - 5.1.1.3 Repeat the Mac Darwin Port Using Absoft FORTRAN Compiler
    - 5.1.2 Install CERESlib
    - 5.1.3 Install SARBLib
  - 5.2 Convert the Main Processor (CER5.1P1)
  - 5.3 Test Main Processor Using Test Suite Software on Target
  - 5.4 Convert the SARB Preprocessors (CER5.0P1)
  - 5.5 Test SARB Preprocessors on the Target Platform

5.6	Convert the SARB HDF Post-processor (CER5.3P1) and Test
5.7	Convert the SARB QC Summary Post-processor (CER5.4P1) and Test
6.0	SARB Detailed Conversion Plan
6.1	Detailed Plan for Library Installation Phase
6.1.1	Detailed Plan for Installing the SDP Toolkit
6.1.1.1	Download the Mac Darwin Distribution Files
6.1.1.2	Install HDF4
6.1.1.3	Install HDF5
6.1.1.4	Install HDF-EOS Version 2
6.1.1.5	Install HDF-EOS Version 5
6.1.1.6	Install the Toolkit
6.1.1.7	Installing the Toolkit Ancillary/Auxiliary (AA) Data Access Tools
6.1.1.8	User Account Setup
6.1.2	Detailed Plan for Installing CERESlib
6.1.3	Detailed Plan for Installing SARBlib
6.2	Detailed Plan for Converting the SARB Main Processor
6.3	Detailed Plan for Testing SARB Main Processor
6.4	Detailed Plan for Converting the SARB Preprocessors
6.5	Detailed Plan for Testing the SARB Preprocessors
6.6	Detailed Plan for Conversion and Test of SARB HDF Post-processor
6.7	Detailed Plan for Conversion and test of SARB QC Summary Processor
7.0	The Conversion Schedule: A Gantt Chart
8.0	Summary

**Listing 1 - Outline of SARB Conversion Plan**

I guarantee that there will be a proportional, positive payback for the amount of effort that you put into your conversion plan. When you write the plan, try to write it from the perspective that someone else might actually have to implement the plan. The following sections discuss the minimum information that should be incorporated in your conversion plan.

## 4.1.1 Document the Subsystem Dependencies



**Mac says 1 The subsystem make files and supporting scripts are the collective key to deducing the external library dependencies, the directory structure model, and the data files that are required to run a PGE on any platform.**

The first subsystem to migrate to the new platform more than likely has the most difficult process. The reason is that all the CERES subsystems are dependent on the PGS Toolkit and the CERES library (CERESlib). Prior to building any subsystem, the Toolkit and CERESlib must be installed and tested. Fortunately, this work was accomplished during the SARB conversion. If you are converting your subsystem to a standalone platform like the Macintosh G5 desktop system, then you may be required to install the Toolkit and CERESlib on the G5 prior to converting your subsystem.<sup>27</sup> For SARB, the third major dependency is the dedicated library, SARBlib, and following the library are the four SARB PGEs. Dependencies common<sup>28</sup> to the subsystem library and the PGEs are the scripts that are employed to prepare for program execution. Such scripts include

<sup>27</sup> See the SARB Conversion Plan for step by step details.

<sup>28</sup> The subsystem library can be treated as though it is a PGE.

make files, staging of data, and the creation of a Process Control File. For example, SARBlib requires only a single make file but *each* SARB PGE requires the following scripts:

- Make file to compile and install the PGS Toolkit message and include files<sup>29</sup>
- Make file to compile and link the program(s) that constitute the PGE<sup>30</sup>
- Make file to compile and link the PGE test suite software
- C-shell script for defining environment variables for the Sampling strategy, Production strategy, the subsystem Configuration codes, and the data sampling date that are employed in the CERES subsystem data-file naming conventions
- C-shell script to generate an ASCII file to define the directory paths and file names that incorporate a complete set of the input and output files for a single execution of a PGE
- C-shell script to generate the Toolkit Process Control File (PCF) using the ASCII file from the previous step as input.
- C-shell script for executing the PGE using the PCF file from the previous step as input
- C-shell script for removing I/O files from the previous run of the PGE in question
- C-shell program for test suite comparison testing

One thing that should be coming increasingly clear is that the subsystem make files and supporting scripts are the collective key to deducing the external library dependencies, the directory structure model, and the data sets that are required to run a PGE on any platform. Evaluate the scripts and the make files for compiler and platform dependencies. For make files, such dependencies will show up as compiler switch settings<sup>31</sup> and library references. The supporting scripts and their outputs typically specify directory path definitions and file names.

In your conversion plan, document each script that is required for building and executing your subsystem components. Walk through each script line by line looking for platform dependencies and references to utilities or special commands that are unique to the source platform<sup>32</sup>. While performing the script walk-through, record all the different directory paths that are referenced in the scripts and make files. In your conversion plan, include a section for documenting the complete directory mapping required by the subsystem. Walking through the scripts and make files will identify most, if not all, of the critical directory paths required for the subsystem. Document<sup>33</sup> each script and each make file that is required for the conversion of the subsystem components. In the conversion plan, specifically call out the changes that must be made to the scripts and make files to get them to run on the target platform. You don't have to call out the exact details for each modification because you may not know what those details are yet; just that it has to change.

---

<sup>29</sup> I lied; SARB only does this once prior to building SARBlib and the four PGEs.

<sup>30</sup> SARB's PGE, CER5.0P1 incorporates 2 executables, each with its own make file.

<sup>31</sup> The IBM XLF compiler will not understand the SGI compiler switches.

<sup>32</sup> The source platform is the platform from which you are converting; in this case, SGI.

<sup>33</sup> In your conversion plan; where else?

### 4.1.2 Document the Subsystem Components

If you have performed a walk-through of each make file and script for the subsystem library(s), each PGE, and the attendant test suite programs, then you should have a pretty good notion of the subsystem component breakdown. In general, the components break down into libraries, individual PGEs, and individual test suite programs. A large PGE with several executables could be further broken down into sub-components if necessary. Document each component in your conversion plan.

The SARB Conversion Plan treated the SARB library separately from the four SARB PGE's, and that is a matter of choice as long as each component is documented in the plan. For each component, if applicable, enumerate the required input files and the expected output files. The file descriptions should include the file name, approximate size in bytes, quantity, and type. For example, the SARB Surface Albedo Monthly Pre-processor requires a maximum of 744 binary Single Satellite CERES Footprint TOA and Surface Fluxes (SSFB) input files. *Each SSFB file can be well over 100 Megabytes in size.* Documenting this type of information in the conversion plan facilitates the formulation of resource requirements and testing practices for the target platform. In this case, a red flag should be waving with regard to the planned approach for staging the SSFB files on the target in preparation for testing the SARB Pre-processor conversion.

### 4.1.3 Document the Subsystem Directory Model

The CERES subsystems appear to share a common directory model starting at the root level of the source platform, where there is a subdirectory named "CERES"<sup>34</sup>. Further, each subsystem appears under the CERES subdirectory. For example, we have /CERES/sarb, /CERES/inversion, /CERES/clouds, /CERES/lib<sup>35</sup>, E.T.C. This pattern appears to continue as each subsystem<sup>36</sup> includes the following subdirectories:

- **lib**, the first step towards the subsystem dedicated library. For SARB, the entire path is /CERES/sarb/lib/src
- **smf**, the Status Message Files that are PGS Toolkit-dependent message objects.
- **rcf**, the Resource Control File subdirectory that further breaks down into:
  - **mcf**, the Metadata Control Files that define the metadata objects for each .met file that is output by the subsystem
  - **pcf**, the Process Control File depository for PCFs that are generated for subsystem PGE executions
  - **PCFgen**, the repository for the ASCII files that are input to the PCF generation scripts
- **bin**, the first step towards the subdirectory containing the PGE executables and other run scripts. For SARB, the entire directory path is /CERES/sarb/bin/sarb.
- **data**, the top level subdirectory leading to input files, output files, Quality Assurance report files, run time log files, and expected output files.
- **test\_suites**, the first step towards the test suite programs for each PGE.

Completely document the directory structure model for your subsystem in the conversion plan. This is the basis of the directory structure that must be created on the target

<sup>34</sup> AKA \$CERESHOME to reference a well-used environment variable (setenv CERESHOME /CERES).

<sup>35</sup> The CERES library, CERESlib, lives at /CERES/lib.

<sup>36</sup> CERESlib being the exception.

platform. If your subsystem references another subsystem for input files, you will also need to define the directory structure for the other subsystem(s) on the target platform. For example, the SARB Monthly Pre-processor references the aforementioned SSFB files at their home in the “inversion” subsystem using the path, /CERES/inversion/data/out\_comp/data. The SARB Monthly Pre-processor can also access data from the Clouds subsystem using the path, /CERES/clouds/data/input/MODIS.

### 4.1.4 Document the Target Platform

The last few sections describe the conversion plan documentation topics for the subsystem as it currently exists on the source platform. Your conversion plan should identify and describe the target platform. This is where we define the target platform resources that we intend to exploit in order to achieve our goal of successfully converting the subsystem in question. We also identify any obstacles that must be overcome to achieve the conversion. Some of the target platform attributes that should be discussed include but are not limited to Big versus Little Endian, Unix to Unix conversion versus Unix to Linux conversion, IEEE compliance on both platforms or not, and compiler availability.

#### 4.1.4.1 The Endian Thing

The SGI source platform is a Big Endian machine. If the target platform is a Little Endian machine<sup>37</sup>, we will need to identify this as a significant aspect of the conversion effort. We also document our analysis of the subsystem components and data files that are impacted by this situation. At this point in the conversion plan we would identify our high level approach to solving the Endian problem. The PPC970 platforms are Big Endian, so there is no Endian problem when converting from SGI to PPC970.

#### 4.1.4.2 Source OS to Target OS

In the conversion plan, the target platform description should address the differences and similarities between the source and the target Operating Systems (OS). Generally, a Unix to Unix conversion will be fairly straight forward but there will be some differences, and if you know what they are, they should be documented in your conversion plan. Section 3 The Macintosh G5 Target Environment identifies some of the differences between the SGI and the Mac G5 Unix environments. If you are converting from Unix to Linux, you will probably discover that there are differences in directory locations for shell tools and some system utilities. For example, attempt to run your subsystem scripts on the target platform. The information that you gain will come in handy when you convert the scripts to run on the target platform.

#### 4.1.4.3 IEEE Compliance

In software conversion efforts, floating point operations always come to the surface. Assess and document your findings on the status of the source and target platforms with regard to mechanisms in use for floating point operations. If the target platform is not IEEE compliant, you will need to formulate an approach for reading binary files containing floating point numbers from the source platform. You will also need to

---

<sup>37</sup> Fortunately, the PPC970 platforms are Big Endian but this is too good of an example to ignore. The Endian problem is so important that it should be addressed even when both platforms are compatible.

formulate an approach to resolving differences in precision for code logic that references floating point variables, and for performing comparisons with the benchmark output data from the source platform. Fortunately, the SGI source platform and the PPC970 target platforms are IEEE compliant.

#### **4.1.4.4 Compiler Availability and Compatibility**

We have already decided that the IBM XL Fortran compiler shall be the compiler of choice for the Mac G5 and the Linux PPC970 platforms (see 3.5 The IBM XL FORTRAN Compiler (IBM XLF)). If your subsystem requires another compiler, then this should be justified and documented in the conversion plan. Selecting an alternative compiler for the target platform may require a significant amount of analysis and test on the target platform.

#### **4.1.5 Document Your High Level Conversion Strategy**

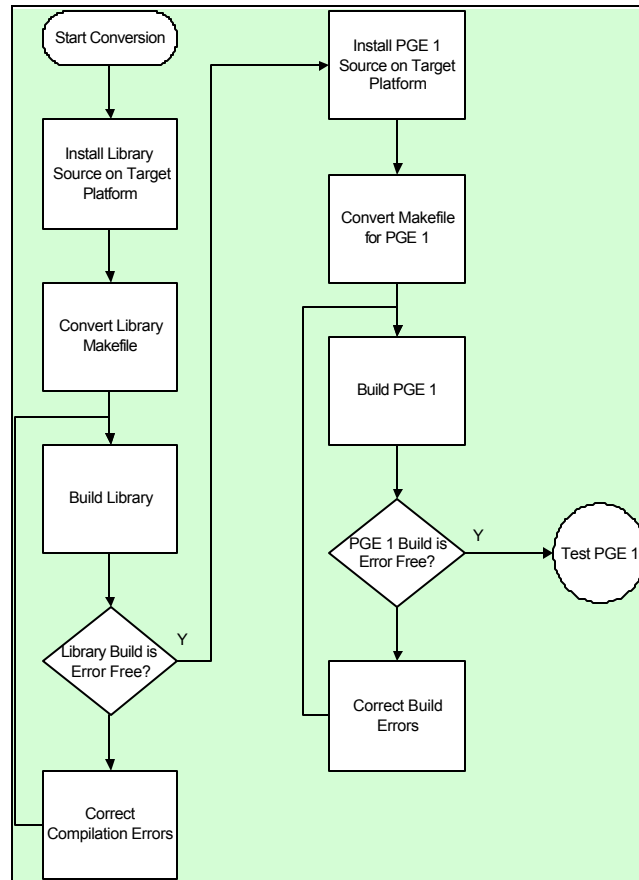
This part of the conversion plan is where you describe your approach to converting the subsystem to the target platform. Provide a high level description of the conversion process, and don't worry about details too much. The next section will describe the detailed approach to this process. Your high level plan should include a process diagram that is similar to the one illustrated in Figure 15.

One of the objectives for converting the SARB subsystem was to prove the feasibility of attaining "10x" processing on the PPC970 platform. Only the Main Processor PGE of the four SARB PGEs was required to run on the target platform to establish the timing measurements that were needed for "10x" feasibility<sup>38</sup>. In that context it seemed prudent to convert the Main Processor PGE first. In contrast, the natural order of conversion of each PGE would have dictated starting with the SARB Monthly Preprocessor PGE followed by the Main Processor PGE. But running the SARB Monthly Preprocessor requires staging several hundred files and requires significant amounts of free disk space on the target platform in the neighborhood of 100 Gigabytes. Thus, the SARB Conversion Plan defined a high level strategy that featured converting and testing the SARB Main Processor first such that timing measurements could be established as quickly as possible.

---

<sup>38</sup> The SARB library had to be installed first since most of the SARB Main Processor subroutines live in the library.





**Figure 15 – Sample Process Diagram**

If you are converting to the Mac G5 platform, does your conversion effort include the PGS Toolkit and CERESlib? If the answer is yes, then your high level description of the conversion process will need to address the Toolkit and CERESlib too. The SARB Conversion Guide incorporates a tested, detailed plan for the Toolkit and CERESlib, so your document can reference the SARB Conversion Plan for those components.

## 4.1.6 Document Your Detailed Conversion Plan



**Mac says 2 - It is a good idea to include a build log for each subsystem library and PGE build sequence.**

This is where the rubber hits the road. The previous section describes how you should document your general approach to converting your subsystem to the target platform. The detailed plan should parallel the high level plan, and it should provide enough guidance that a junior software engineer should be able to implement the conversion on the target platform. For the subsystem library(s), each PGE, and each test suite program, the detailed plan should provide the following:

- A table of all the environment variables that must be defined prior to building and executing the component

- A complete sequence of target system commands for installing the component source code
- A complete sequence of target system commands for installing the component data files
- A complete sequence of commands that invoke the target-converted make file or script for compiling and installing the subsystem message and include files
- A complete sequence of commands that invoke the target-converted make file or script for building the component
- A complete sequence of commands that invoke the target-converted make file or script for executing the component<sup>39</sup>
- A complete sequence of commands that invoke the target-converted make file or script for building the component test suite
- A complete sequence of commands that invoke the target-converted make file or script for executing the component test suite programs

Section 6 of The SARB Conversion Plan divided the detailed conversion process into two subsections per PGE. The first subsection for a PGE should describe the sequence of commands to install and build the source code on the target<sup>40</sup>, and the second subsection should describe the sequence of commands needed to execute a test case<sup>41</sup>. If data files were required for the test case, the second section of the pair would provide the commands steps needed to install the input files if they had not been loaded in a previous step. It is a good idea to include a build log for each subsystem library and PGE build sequence. As an example of the level of detail and the creation of a build log, Listing 2 is an excerpt from Section 6.1.3 of the SARB Conversion Plan:

**Compile the SARB library. Make sure you convert the make file first. For example, Makefile.CRS incorporates a library command that is not compatible with Darwin. The SGI version of the command is “ar -rf”, and for Darwin it should be “ar -rs”:**

```
o cd $CERESHOME/sarb/lib/src
o script buidSARLib.log
o make -f Makefile.CRS clean
o make -f Makefile.CRS
o exit
```

**Listing 2 - Excerpt from SARB Conversion Plan**

Note that the details for converting Makefile.CRS are left to the programmer but the steps for building the library are called out explicitly, including the creation and termination (exit) of the log file for the build. The build log will come in handy during testing if you want to examine how a certain module was compiled, to determine if a certain module had compilation warning messages, or what libraries were included in the link step. Also, the build log is very handy if there are lots of compilation errors and warnings during the build.

<sup>39</sup> Obviously, you would not be able to execute a library but you get the point.

<sup>40</sup> For the SARB Conversion Plan this included the test suite software.

<sup>41</sup> For the SARB Conversion Plan this included the test suite execution and file comparisons using the Unix diff utility.

When the conversion plan is implemented, be sure to update the plan document to include command steps that were added or eliminated during the process. This is important because you may perform this conversion more than once or twice.<sup>42</sup>

#### 4.1.7 Estimate and Document Your Subsystem Conversion Schedule



**Mac says 3 - I think it is a good idea to lump your contingency time following each major phase in the schedule rather than allotting contingency time after every subtask in the schedule.**

Prepare a Gantt chart that describes your estimated resources and time required to complete the conversion. Use the detailed conversion plan subsections to lay out your conversion tasks. If more than one person is implementing your conversion plan, then show parallel activities in your Gantt chart. While you are thinking through the process and considering the time line, you may decide to revise your conversion plan. This is probably a good thing, especially if you are adding in more details. Don't forget to schedule in contingency time in your Gantt chart<sup>43</sup>. I think it is a good idea to lump your contingency time following each major phase in the schedule rather than allotting contingency time after every subtask in the schedule.

Preparing a detailed schedule is an important exercise as it causes you to identify possible problems that might cause delays in your conversion progress. For example, the SARB Conversion Plan included performing a timing analysis on the new DACC cluster. When the SARB Conversion Plan was written, the hardware components for the new cluster had not yet been ordered from the manufacturer. Estimating the time required for getting the SARB Main Processor PGE up and running on the Mac G5 platform identified the earliest possible milestone date for having the new cluster available for timing studies. Assuming that the estimate for possible contingencies was accurate, allowed for the estimation of a preferred window of availability for the DAAC cluster.<sup>44</sup> At a point like this in your schedule estimation you might realize that it may be prudent to have a plan B available in case a critical resource is not available when you estimate that you will need it. Your Plan B should be documented in the conversion plan in case you need to implement it.

#### 4.2 Work Your Plan

Hopefully, you put a righteous effort into writing a first-rate conversion plan because now it is time to work your plan. Print out a copy of the section that defines the detailed conversion plan. Assemble the conversion team and provide a copy of the detailed plan to each person on the team. Assign each member of the conversion team one or more

---

<sup>42</sup> I'm not kidding. During the SARB conversion to the Mac G5 platform, I had to re-install the Toolkit about 5 times, CERESlib about 6 times, and the SARB subsystem 3 times. Each and every time I was very thankful that I kept the SARB Conversion Plan updated with lessons learned from previous installations!

<sup>43</sup> During the SARB conversion, my target platform went down for several weeks, and that ate up almost all of my contingency time.

<sup>44</sup> In reality, the DAAC cluster missed the window and the new cluster is not available as I write this footnote!

sub-sections, and get them started. Hopefully, in your schedule you accounted for parallel tasks. If your subsystem has a library, there is no doubt that the library will need to be completely installed first<sup>45</sup>, so you may want to stagger the starting dates for the various members of the conversion team.<sup>46</sup> Unless you have more than one target platform, it is probably a good idea to limit the size of the conversion team to two or three people.<sup>47</sup> Perhaps one person can be limited to performing the necessary tasks on the source platform. During component testing you may find that you are running two or three test cases on the source platform per day to get intermediate data results for comparison with the results coming from the target platform. Also, your plan probably calls out details for installing and building source code on the target platform, and this assumes the existence of one or more delivery packages that are available on the source platform. If the data files that constitute a complete test case for your subsystem are significant in terms of storage requirements, then you may have called out a separate delivery package for the data files. In fact, you may have a separate source code and data file delivery package for each PGE. The bottom line here is that your detailed plan should identify 1) a compressed tar file, 2) a series of steps for electronic transfer of files from the source platform to the target platform, or 3) the necessary steps for staging data files to an external disk that is compatible with the target platform. The third alternative implies that your detailed plan specifies the sequence of steps necessary to transfer the source platform files onto the external disk in addition to the process for interfacing the external disk with the source platform and finally with the target platform. The logistics to the 3<sup>rd</sup> alternative may be limited by your access to the source platform and the availability of an external disk that is compatible with both the source and the target platforms.

### 4.3 Library Install Example



**Mac says 4 - The CERESlib delivery package includes an extensive test suite, and ideally, every subsystem library should have an attendant test package that is platform independent.**

When I read a “how to” book I always look for the examples to see if I understand what the “how to” text is trying to say. Since a typical subsystem probably has its own library, it makes sense to provide an example that includes the preparation work that occurs on the source platform as well as the work that occurs on the target platform. When I was considering what to use for the library install example, I was tempted to use the CERESlib port<sup>48</sup> from the SGI platform to the Macintosh G5 platform. The CERESlib delivery package includes an extensive test suite, and ideally, every subsystem library should have an attendant test package that is platform independent. What does that mean? Platform independent means that the test code would be written in strict

<sup>45</sup> Don’t forget to include the PGS Toolkit and CERESlib.

<sup>46</sup> For the SARB conversion, I had a team of one, so there were no parallel tasks.

<sup>47</sup> The Mac G5 can handle multiple users using Secure Shell logins at the Darwin command line level.

<sup>48</sup> I’m deliberately using the term “port” here because CERESlib is well packaged and the library itself requires very few conversion modifications. The CERESlib test suite software did require a conversion since it tests the SGI compilers.

adherence to the CERES language standard of choice. For CERESlib the standard would be Fortran 90 or Fortran 95. Furthermore, the test code should test the library modules functionally, avoiding any platform-specific code. Sometimes platform specifics can not be avoided and the current CERESlib code-level documentation successfully attempts to identify all such cases. But, in reality, most of the subsystem dedicated libraries probably do not include a comprehensive test suite, if they have one at all. This was the case for SARBlib. The implication is that you get to test the library code when you are trying to test the downstream PGEs. I have selected the SARBlib installation because it is probably more realistic, and I know the example has been tested.

### 4.3.1 Create a Delivery Package on the Source Platform

You are probably scoffing at this example since you may already have done this work to deliver your library code to the DAAC. If that is the case, you can skip this example. We are converting from a Unix platform to either a Unix or Linux platform. The Unix *tar* utility works wonderfully on most Linux systems, so it is my choice for archiving software files written in any language. The nice thing about the *tar* utility is that it preserves the directory structure so we can port the source code and the supporting directory structure in the same operation. In this way we can save some effort in creating the directory structure on the target machine. Even though this example is a library port<sup>49</sup>, let's make it realistic and port all the subsystem source code just the way I did it when I converted the SARB subsystem to the Mac G5.

To start our example we log on to the source platform, and change directories to the subsystem, home directory under the CERES home directory (\$CERESHOME). I won't use environment variables in our example. Start creating a "tarball":

- `cd /CERES/sarb`
- `tar -cvf ~/SARB_SW.tar lib/src/*50`
- `tar -uvf ~/SARB_SW.tar src/sarb/mainss5/*51`
- `tar -uvf ~/SARB_SW.tar src/sarb/press5_monthly/*52`
- `tar -uvf ~/SARB_SW.tar src/sarb/press5_modisaer/*53`
- `tar -uvf ~/SARB_SW.tar src/sarb/hdf2crsb/*54`
- `tar -uvf ~/SARB_SW.tar src/sarb/crsb_check/*55`

<sup>49</sup> There he goes again, using that word "port". Since we have no test suite, all we can do is port the code rather than convert the code. We will convert the library code "on the fly", AKA "the hard way".

<sup>50</sup> Two things; 1) I don't have the proper permissions to create a tar file in /CERES/sarb so I write the tar file in my home directory on the target, and 2) I use \* as opposed to \*.\* because \*.\* will not get files without a file suffix, and because I knew the library has no .obj and .mod files. If it did contain files that I don't want in my tar file, I would be forced to invoke *tar* successive times using \*.f\* and Make\*

<sup>51</sup> This updates our now existing tar file with the SARB Main Processor PGE (CER5.1P1). What really happens in this step is that I find out that there is a subdirectory (src/sarb/mainss5/crs\_hdf) for which I do not even have "read" permissions. So, in the real life case, I had to port this software separately after contacting the SARB subsystem lead to either modify the permissions or provide me with a tar file.

<sup>52</sup> This is one of the two executables that constitute the SARB Monthly Preprocessors for the PGE, CER5.0P1.

<sup>53</sup> This is the 2<sup>nd</sup> of the two executables that constitute the SARB Monthly Preprocessors for the PGE, CER5.0P1.

<sup>54</sup> One of the 3 directories for CER5.4P1. This is the source for a program that converts HDF files to their CRSB equivalent in preparation for comparison with the original CRSB file.

<sup>55</sup> One of the 3 directories for CER5.4P1. This is the source for a program that compares two CRSB files for equality.

- `tar -uvf ~/SARB_SW.tar src/sarb/qc_check/*`<sup>56</sup>
- `tar -uvf ~/SARB_SW.tar test_suites/sarb/*`<sup>57</sup>
- `tar -uvf ~/SARB_SW.tar smf/sarb/*`<sup>58</sup>
- `tar -uvf ~/SARB_SW.tar bin/sarb/*`<sup>59</sup>
- `tar -uvf ~/SARB_SW.tar rcf/*`<sup>60</sup>
- `cd ~`
- `compress SARB_SW.tar`

Now we have a source delivery “tarball”. I compressed it because in this example we are working with a 3-Megabyte tar file. Use your favorite file transfer protocol to transfer the compressed tar file to the target platform.

### 4.3.2 Install the Subsystem Source on the Target Platform

Recall that we are using SARB as a basis for the library install example. We have most of the SARB source code on a compressed tar file in the home directory on the target platform. Before we perform a tar extraction, we need to prepare the top of the subsystem directory structure such that we have owner and group permissions in all our subsystem subdirectories. For the CERES home directory, we can have the root own the CERES subdirectory and give ourselves group permissions. We will allow owner and group to read, write, and execute, and we will allow the world to read and execute but not to write. If this is not the case, then make it so:

- `cd /`
- `su root`
- `chgrp donaldsn CERES`
- `chmod 775 CERES`
- `exit`

Our example is within the SARB context so we will now create the SARB home directory:

- `cd /CERES`
- `mkdir sarb`
- `chmod 775 sarb`

Since the group has write permission in the CERES subdirectory, then user donaldsn can create the sarb subdirectory under the CERES directory. The /CERES/sarb directory will have donaldsn as the owner and as the group. You may choose to have owner and group be different, so give both owner and group read, write, and execute permissions but let the world have only read and execute permission. In my example donaldsn is both owner and group. Now, we can extract our software in the /CERES/sarb subdirectory and the

---

<sup>56</sup> One of the 3 directories for CER5.4P1. This is the source for the SARB QC Summary Processor.

<sup>57</sup> Only CER5.0P1 and CER5.1P1 have test suite code.

<sup>58</sup> I always forget the Status Message Files but not here. They are the 1<sup>st</sup> step in the source build.

<sup>59</sup> Most of the SARB scripts are stored in /CERES/sarb/bin/sarb. The scripts that are stored elsewhere have been collected in the previous tar updates.

<sup>60</sup> This captures the rcf directory structure and the source files stored in /CERES/sarb/rcf/mcf.

file permissions will be good throughout the whole directory hierarchy. Continuing with our example, we now install our source code on the target platform:

- `cd ~`
- `mv SARB_SW.tar.Z /CERES/sarb`
- `cd /CERES/sarb`
- `uncompress SARB_SW.tar.Z`
- `tar -xvf SARB_SW.tar`
- `compress SARB_SW.tar`

We have one more thing to do with regard to the directory structure in preparation for building the library. We need to establish directory locations for the Toolkit Message and Include files. Here is what is needed:

- `cd /CERES/sarb`
- `mkdir PGS`
- `mkdir PGS/include`
- `mkdir PGS/message`

That's it for installing the software and defining the directory structure for most, if not all, of the source code. We have really accomplished a lot in that we have most of the source code on the target, and the source file directory structure has been transferred to the target platform with a minimum of effort. We can proceed to the library build.

### 4.3.3 Build the Library

Our library build example assumes that the PGS Toolkit and CERESlib have already been installed on the target platform. One of the by-products from the Toolkit and CERESlib installations is a C-shell script named "ceres-env.csh". For performance of SARB conversion work on the target, it is convenient to source `ceres-env.csh` each time you open a terminal window (shell). Put the statement, "`source ceres-env.csh`" in your `.cshrc` file which is hidden in your home directory. I have listed `ceres-env.csh` here for convenience:

```
#####
#
# Name: ceres-env.csh
#
# Purpose: This script sets up environment variables
#          needed by CERES subsystems during code
#          compilation and execution.
#
# Target platform: Mac G5
#
# Target compiler: IBM XL Fortran
#
# compilation mode: regular
#
#####

set brand = macintosh
setenv CERES_STARTUP_SCRIPT ceres-env.csh

#-----#
# PGS Toolkit directory #
#-----#
setenv HDF5INC "."
setenv PGSDIR /opt/net/TOOLKIT
source $PGSDIR/bin/$brand/pgs-dev-env.csh
```

```

setenv PGS LIB $PGSDIR/lib/$BRAND
set path = ($path $pgs_path)
#-----#
# HDF library information #
#-----#
setenv HDF5DIR      "-L$PGSDIR/hdf5/$BRAND/hdf5-1.6.1/lib"
setenv HDF5LIBS     "-lhdf5"
setenv HDFEOS5DIR   "-L$PGSDIR/hdfeos5/lib/$BRAND"
setenv HDFEOS5LIB   "-lgctp -lhe5_hdfeos"
setenv LD_LIBRARY_PATH $PGSDIR/hdf5/$BRAND/hdf5-1.6.1/lib

setenv HDFDIR      "-L$PGSDIR/hdf/$BRAND/HDF4.2r0/lib"
setenv HDFLIBS     "-lmfhdf -ldf"
setenv HDFEOSDIR   "-L$PGSDIR/hdfeos/lib/$BRAND"
setenv HDFEOSLIB   "-lhdf5 -lgctp"
setenv ADD_LFLAGS   "-L/usr/local/jpeg-6b/lib -L/usr/local/zlib-1.1.4/lib"
setenv ADD_LIBS     "-ljpeg -lz"

#-----#
# CERES home and CERESLIB directories #
#-----#
setenv CERESHOME /CERES
setenv CERESLIB $CERESHOME/lib

#-----#
# Directory for Toolkit-accessible CERES constants file #
#-----#
setenv PGSCONSDIR $CERESLIB/data

#-----#
# Compiler locations and default flags #
#-----#
setenv CC          'gcc'
setenv CFLAGS      '-c -DMACINTOSH'
setenv XLF          /opt/ibmcmp/xlf/8.1/bin/xlf90
setenv CLOAD
setenv F90          $XLF
setenv F90COMP      '-qxlf90=nosignedzero,autodealloc -O2 -qmaxmem=32768 -c -qextname -
qsuffix=f=f90'

setenv FCOMP        '-qxlf90=nosignedzero,autodealloc -O2 -qmaxmem=32768 -c -qextname -
qfixed=80'

#setenv F90LOAD      '-static -s'

setenv F90LOAD
setenv F90LIBDIR     '/opt/ibmcmp/xlf/8.1/lib'
setenv F90LIB        $F90LIBDIR/libxlf90.dylib

#-----#
# CERES global include and message directories #
#-----#
setenv PGSMMSG /CERES/sarb/PGS/message
setenv PGSINC /CERES/sarb/PGS/include

#-----#
# path cleanup #
#-----#
if (-e $PGSINC) diff $PGSINC/PGS_SMF.h $PGSDIR/include/PGS_SMF.h >& /dev/null
if ($status != 0) then
    $CERESLIB/bin/cp_inc_and_msg_files.csh
endif
source $CERESLIB/bin/cleanpath

```

Listing 3 - CERES Environment Variable Definition Script

I listed ceres-env.csh, C-shell script above so I could present the SARBLib make file, Makefile.CRS, at this point:



## CERES Conversion Guide

```

PROG = SARBlib_CRS.a

SRCS = Ancill_Init.f90 Collins_assimilation.f90 Constrain_Params.f90 \
Control_Mod.f90 Convert_OptDepth.f90 DrivIngest.f90 DrivTab_Var.f90 \
FLSA_LUT_Uutils.f90 FL_IO_Params.f90 FL_Pass_Interface.f90 \
FL_SetUp.f90 GADS_Aer.f90 IGBP_AdjSnowIce.f90 IGBP_Uutils.f90 \
Lev_Isolate.f90 MonMODISAer_ErrParams.f90 MonMODISAer_Params.f90 \
Monthly_AerHist_Uutils.f90 Monthly_SfcAlb_IO.f90 No_Cloud.f90 \
Profile_Params_CRS.f90 QC_Accum.f90 QC_Fin.f90 QC_Init.f90 \
SARBAer_Uutils.f90 SARBAer_Var.f90 SARBInput_Params.f90 \
SARBInput_Uutils.f90 SARB_Error_Process.f90 SARB_FOV_Albedo.f90 \
SARB_General.f90 SARB_IO_Uutils.f90 SARB_OutVar.f90 \
SARB_QC_Params_CRS.f90 SARB_Var.f90 SigTab_Var.f90 SigmaIngest.f90 \
Spectral_Dat.f90 Spectral_Sfc.f90 TuneDrive.f90 Tune_Code.f90 \
UpTropHum.f90 VMax_Min.f90 With_Cloud.f90 ZJIN_Mod.f90 \
ZJIN_Params.f90 alblib_data.f90 aotfit.f90 extras.f90 fuinput.f90 \
fuoutput.f90 fuprint.f90 gfdl_aer_clim.f90 ma_tip.f90 \
match_profiles.f90 rad_multi_0403.f90 sei_ji_k2b.f90 \
sei_ji_solver_0403.f90 sfcAlb_history.f90 sktbl_ht02a.f90 taucorr.f90 \
uvcor_all.f90 wssacomp.f90

OBJS = Ancill_Init.o Collins_assimilation.o Constrain_Params.o Control_Mod.o \
Convert_OptDepth.o DrivIngest.o DrivTab_Var.o FLSA_LUT_Uutils.o \
FL_IO_Params.o FL_Pass_Interface.o FL_SetUp.o GADS_Aer.o \
IGBP_AdjSnowIce.o IGBP_Uutils.o Lev_Isolate.o MonMODISAer_ErrParams.o \
MonMODISAer_Params.o Monthly_AerHist_Uutils.o Monthly_SfcAlb_IO.o \
No_Cloud.o Profile_Params_CRS.o QC_Accum.o QC_Fin.o QC_Init.o \
SARBAer_Uutils.o SARBAer_Var.o SARBInput_Params.o SARBInput_Uutils.o \
SARB_Error_Process.o SARB_FOV_Albedo.o SARB_General.o SARB_IO_Uutils.o \
SARB_OutVar.o SARB_QC_Params_CRS.o SARB_Var.o SigTab_Var.o \
SigmaIngest.o Spectral_Dat.o Spectral_Sfc.o TuneDrive.o Tune_Code.o \
UpTropHum.o VMax_Min.o With_Cloud.o ZJIN_Mod.o ZJIN_Params.o \
alblib_data.o aotfit.o extras.o fuinput.o fuoutput.o fuprint.o \
gfdl_aer_clim.o ma_tip.o match_profiles.o rad_multi_0403.o \
sei_ji_k2b.o sei_ji_solver_0403.o sfcAlb_history.o sktbl_ht02a.o \
taucorr.o uvcor_all.o wssacomp.o

OBJS_F = WindowFilter.o aerosols_0403.o aqua_wnflt_0404.o chou_routines.o \
misc_0403.o sei_ji_twostreamsolv_sw_v20.o

TKLIBS = -L$(PGSLIB) -lPGSTK
CLIBS = $(CERESLIB)/data_products.a $(CERESLIB)/cereslib.a
MOD_FLAG = -I. -I$(CERESLIB)/mod
LIBS =

INC_FLAG = -I$(PGSINC) -I$(HDFINC) -I$(HDF5INC) -I../include -I./include
VPATH = .:$(PGSINC):$(HDFINC):$(HDF5INC):../include:./include

HDF_FLAGS = $(HDFEOSDIR) $(HDFEOSLIB) $(HDFDIR) $(HDFLIBS)
HDF5_FLAGS = $(HDFEOS5DIR) $(HDFEOS5LIB) $(HDF5DIR) $(HDF5LIBS)
DAAC_FLAGS = $(ADD_LFLAGS) $(ADD_LIBS)

#CC =
#CFLAGS =
#F90 =
F90COMP = -O2 -64 -c
FCOMP = -O2 -w -64 -c
#F90LOAD =

all: $(PROG)

$(PROG): $(OBJS) $(OBJS_F)
        ar rf $@ $?
#       -\mv *.mod ../mod
#       \cp $@ ../lib

clean:
        -\rm -f $(PROG) $(OBJS) $(OBJS_F)
        -\rm -f *.mod

.SUFFIXES:

.SUFFIXES: .f90 .mod .f .c .o

```

```

.f90.o:
    $(F90) $(F90COMP) $(MOD_FLAG) $(INC_FLAG) $<

.f90.mod:
    $(F90) $(F90COMP) $(MOD_FLAG) $(INC_FLAG) $<

.f.o:
    $(F90) $(FCOMP) $(MOD_FLAG) $(INC_FLAG) $<

.c.o:
    $(CC) $(CFLAGS) $(MOD_FLAG) $(INC_FLAG) $<

Ancill_Init.o: Collins_assimilation.o SARBInput_Params.o SARB_Var.o \
    gfdl_aer_clim.o PGS_ANCINIT_25725.f
Collins_assimilation.o: SARBAer_Var.o SARB_IO_Utills.o
Constrain_Params.o: Profile_Params_CRS.o
Control_Mod.o:
Convert_OptDepth.o: FL_IO_Params.o Profile_Params_CRS.o SARBInput_Params.o \
    SARB_Var.o
DrivIngest.o: DrivTab_Var.o SARB_Error_Process.o SARB_Var.o
FLSA_LUT_Utills.o: PGS_FLSALUTIO_25724.f
FL_IO_Params.o: Profile_Params_CRS.o
FL_Pass_Interface.o: FL_IO_Params.o MonMODISAer_Params.o Profile_Params_CRS.o \
    SARBAer_Var.o SARBInput_Params.o SARB_Var.o match_profiles.o \
    rad_multi_0403.o
FL_SetUp.o: Convert_OptDepth.o FL_Pass_Interface.o Profile_Params_CRS.o \
    SARB_Error_Process.o SARB_OutVar.o SARB_QC_Params_CRS.o SARB_Var.o \
    TuneDrive.o UpTropHum.o With_Cloud.o
GADS_Aer.o: PGS_GADSAER_25715.f
IGBP_AdjSnowIce.o: FL_IO_Params.o IGBP_Utills.o SARBInput_Params.o
IGBP_Utills.o: PGS_IGBPUTIL_25721.f
MonMODISAer_Params.o:
Monthly_AerHist_Utills.o: MonMODISAer_ErrParams.o MonMODISAer_Params.o \
    SARBInput_Params.o SARB_Error_Process.o SARB_IO_Utills.o SARB_Var.o \
    aotfit.o
Monthly_SfcAlb_IO.o: sfcalb_history.o PGS_MSFCALBIO_25722.f
No_Cloud.o: Profile_Params_CRS.o SARB_General.o SARB_OutVar.o SARB_Var.o
QC_Accum.o: FL_IO_Params.o Profile_Params_CRS.o SARBInput_Params.o \
    SARB_OutVar.o SARB_QC_Params_CRS.o SARB_Var.o
QC_Fin.o: Profile_Params_CRS.o SARB_OutVar.o SARB_QC_Params_CRS.o SARB_Var.o
QC_Init.o: Profile_Params_CRS.o SARB_QC_Params_CRS.o
SARBAer_Utills.o: Collins_assimilation.o FL_IO_Params.o MonMODISAer_Params.o \
    SARBAer_Var.o SARBInput_Params.o SARB_OutVar.o SARB_Var.o \
    gfdl_aer_clim.o wssacomp.o
SARBInput_Params.o:
SARBInput_Utills.o: FL_IO_Params.o Profile_Params_CRS.o SARBInput_Params.o \
    SARB_OutVar.o SARB_Var.o
SARB_Error_Process.o: MonMODISAer_ErrParams.o SARB_Var.o \
    PGS_DERIVLOAD_25709.f PGS_FLMODEL_25706.f PGS_FLXRANGE_25705.f \
    PGS_INGEST_25702.f PGS_SFICALBCALC_25707.f PGS_SIGMALOAD_25703.f \
    PGS_TUNEDRV_25704.f
SARB_FOV_Albedo.o: FL_IO_Params.o Profile_Params_CRS.o SARB_Var.o \
    Spectral_Sfc.o
SARB_General.o: FL_IO_Params.o VMax_Min.o
SARB_IO_Utills.o: PGS_SARBIOUTIL_25750.f
SARB_OutVar.o: Profile_Params_CRS.o
SARB_QC_Params_CRS.o: Profile_Params_CRS.o
SARB_Var.o: Profile_Params_CRS.o
SigmaIngest.o: SARB_Error_Process.o SARB_Var.o SigTab_Var.o
Spectral_Sfc.o: FLSA_LUT_Utills.o FL_IO_Params.o IGBP_Utills.o \
    Profile_Params_CRS.o SARBInput_Params.o SARB_OutVar.o SARB_Var.o \
    Spectral_Dat.o ZJIN_Mod.o sfcalb_history.o
TuneDrive.o: Constrain_Params.o Convert_OptDepth.o FL_IO_Params.o \
    Profile_Params_CRS.o SARBInput_Params.o SARB_Error_Process.o \
    SARB_General.o SARB_QC_Params_CRS.o SARB_Var.o Tune_Code.o
Tune_Code.o: Control_Mod.o DrivTab_Var.o SARBInput_Params.o \
    SARB_Error_Process.o SARB_Var.o SigTab_Var.o
UpTropHum.o: SARBInput_Params.o
With_Cloud.o: FL_IO_Params.o Profile_Params_CRS.o SARB_General.o
ZJIN_Mod.o: ZJIN_Params.o
fuinput.o: Profile_Params_CRS.o taucorr.o
fuoutput.o: fuinput.o
fuprint.o: fuinput.o fuoutput.o

```

```

gfdl_aer_clim.o: PGS_GFDLAER_25716.f
ma_tip.o: fuinput.o
match_profiles.o: FL_IO_Params.o Profile_Params_CRS.o SARBInput_Params.o \
    SARB_IO_Utills.o SARB_Var.o
rad_multi_0403.o: fuinput.o fuoutput.o uvcor_all.o
seiji_k2b.o: fuinput.o seiji_k2b.o sktbl_ht02a.o
seiji_solver_0403.o: fuinput.o fuoutput.o
sfcalb_history.o: IGBP_Utills.o
uvcor_all.o: fuinput.o fuoutput.o uvcor_all.o

```

Listing 4 - SARBlib Makefile Script

I have highlighted in red the lines that need to be modified to make this make file script ready to run on the target platform. The make file provides a way to override the compiler switches that are set in `ceres-env.csh`. This can be convenient, so I would just delete the SGI compiler switch syntax and comment out those two lines (F90COMP and FCOMP definitions) for later usage. The 3<sup>rd</sup> modification is to:

change `ar rf $@ $?` to `ar rs $@ $?`

because the *ar* switches are different on Darwin. That's it for the make file.

Before we can build the library, we need to compile the subsystem Status Message Files (SMF). The SARB subsystem compiles the SMF files with a make file but I think it is much more straight forward to do it with a C-shell script that can be acquired (stolen?) from CERESlib. The CERESlib script is named `smfcompile_all.csh`, and I have listed it here for convenience:

```

#!/bin/csh -f
#####
#
# !csh
# Name: smfcompile_all.csh
#
# !Description
# Routine ID:
#
# Purpose:
# This script does an smf compile of all *.t files in the local
# directory. It then moves the created include and message files
# to the appropriate directories.
#
# Command-line Parameters:
# $1: mvToPGS: Set this value to anything other than "y" or "Y"
#           if you do not want the created include and message
#           files do get copied to the PGSINC and PGSMMSG
#           directories.
#           (This option should only be used from the CERESlib account)
#
# Return Values:
# none
#
# !Team-Unique Header:
#
# Notes:
#
# !Revision History:
# Revision 1.1 1999/11/16
# Joe Stassi, SAIC (j.c.stassi@larc.nasa.gov)
# 1. Modified code to be able to turn off copy to PGSINC and PGSMMSG
#    directories.
# 2. Added use of echo_string and border_echo scripts.
#
# Revision 1.1 1998/06/24
# Joe Stassi, SAIC (j.c.stassi@larc.nasa.gov)
# Initial version of code
#

```

```

# !end
#
#####

set border_echo = $CERESLIB/bin/border_echo.csh
set echo_string = $CERESLIB/bin/echo_string.csh

set errcount = 0

if ($#argv >= 1) then
    set mvToPGS = $1
else
    set mvToPGS = "Y"
endif

#-----
# smf compile each *.t file
#-----
foreach file ( *.t )

    $PGSBIN/smfcompile -f $file -f77
    if ($status != 0) @ errcount ++

    $PGSBIN/smfcompile -f $file
    if ($status != 0) @ errcount ++

end

#-----
# For CERESlib account
#-----
#-----
# copy include and message files to include and message directories
#-----
if ($USER == cerlibcm) then

    echo ""
    $echo_string "Copying .f files to CERESlib include directory"
    \cp -f *.f $CERESLIB/include
    if ($status == 0) then
        echo "okay"
    else
        echo "ERROR"
        @ errcount ++
    endif

    $echo_string "Copying .h files to CERESlib include directory"
    \cp -f *.h $CERESLIB/include

    if ($status == 0) then
        echo "okay"
    else
        echo "ERROR"
        @ errcount ++
    endif

    $echo_string "Copying PGS message files to CERESlib message directory"
    \cp -f PGS* $CERESLIB/message
    if ($status == 0) then
        echo "okay"
    else
        echo "ERROR"
        @ errcount ++
    endif

endif

#-----
# For all accounts (unless $1 exists and is != "Y")
#-----
#-----
# Move include and message files to PGSINC and PGMSG directories
#-----
if ($mvToPGS =~ [Yy]) then

```

```

echo ""
$echo_string "Moving .f files to PGSINC directory"
\mv -f *.f $PGSINC
if ($status == 0) then
    echo "okay"
else
    echo "ERROR"
    @ errcount ++
endif

$echo_string "Moving .h files to PGSINC directory"
\mv -f *.h $PGSINC
if ($status == 0) then
    echo "okay"
else
    echo "ERROR"
    @ errcount ++
endif

$echo_string "Moving PGS message files to PGSMMSG directory"
\mv -f PGS* $PGSMMSG
if ($status == 0) then
    echo "okay"
else
    echo "ERROR"
    @ errcount ++
endif

else
    \rm PGS*
endif

#-----
# indicate SUCCESS or FAILURE
#-----
echo ""
if ($errcount == 0) then
    $border_echo "SUCCESS SUCCESS SUCCESS SUCCESS SUCCESS SUCCESS SUCCESS"
else
    $border_echo "PROBLEMS PROBLEMS PROBLEMS PROBLEMS PROBLEMS PROBLEMS"
endif
echo ""

```

Listing 5 - Build Script for Status Message Files

I have highlighted in red the script lines that should be removed for usage with our subsystem. Store the converted smfcompile\_all.csh file in /CERES/sarb/smf/sarb. Finally, we can proceed with the preparations for building the library:

- source ceres-env.csh
- cd \$CERESHOME/sarb/smf/sarb
- script buildSARBsmf.log
- smfcompile\_all.csh
- \$CERESLIB/bin/cp\_inc\_and\_msg\_files.csh
- exit

If we got through the SMF build with no errors we can now build the library:

- cd \$CERESHOME/sarb/lib/src
- script buildSARBlb.log
- make -f Makefile.CRS clean
- make -f Makefile.CRS
- exit

Chances are there will be some compilation errors and warnings so the library build will be an iterative process. Since there is no test suite, you are ready to move on to the first PGE conversion when you have successfully created the library.

## 4.4 Build PGE and Test Example

From a conversion perspective, the most difficult SARB PGE to convert was the SARB Main processor, CER5.1P1. For this example, I will walk us through a build of the SARB Main processor and the subsequent execution of a test case that causes a signal trap on an invalid floating point operation. The compiler configuration for the example is the one discussed in section 3.5.4 Complete Build and Sweep for Invalid Floating-Point Operations. This example assumes that the source code has already been ported and installed, and that the initial sweep for compiler errors and warnings has been completed. Also, assumed is that the test case data files have been ported and installed in the appropriate subdirectories.

### 4.4.1 Build the PGE

Recall that in section 4.3.3 Build the Library, I listed the C-shell script that defines all the necessary environment variables for compiling, linking, and executing CERES PGEs. To configure for trapping invalid floating point operations, we need to modify the F90COMP and FCOMP environment variables to reconfigure the IBM XLF compiler<sup>61</sup>. So, for our example, we modify ceres-env.csh to set F90COMP and FCOMP as follows:

```
setenv F90COMP '-qinitauto=7fbffff -qfloat=nans -qflttrap=zero:inv:en -qsigtrap -qlist -
qsource -qxlf90=nosignedzero,autodealloc -O2 -qmaxmem=32768 -c -qextname -qsuffix=f=f90'
setenv FCOMP '-qinitauto=7fbffff -qfloat=nans -qflttrap=zero:inv:en -qsigtrap -qlist -
qsource -qxlf90=nosignedzero,autodealloc -O2 -qmaxmem=32768 -c -qextname -qfixed=80'
```

The SARB Main processor depends heavily on SARBlib so we must re-build SARBlib for the test:

- source ceres-env.csh
- cd \$CERESHOME/sarb/lib/src
- script buildSARBlib.log
- make -f Makefile.CRS clean
- make -f Makefile.CRS
- exit

At this point the SARB library is configured for trapping invalid floating point operations. Now we must re-build the SARB Main processor in preparation for the test:

- cd \$CERESHOME/sarb/src/sarb/mainss5
- script buildSARBmain.log
- make clean
- make
- exit

---

<sup>61</sup> You don't have to do it this way. You can redefine them on the command line, or you can create a separate script, or you can modify the various make files associated with the PGE in question. However, this testing phase may take days or even weeks to complete. I like to have the current configuration set up implicitly each time I open a new terminal window.

At this point we are completely configured for trapping invalid floating point operations, and we can configure for the test and execute it.

#### 4.4.2 Test the PGE

We are not quite ready to execute the test case. The CERES subsystems are PGS Toolkit-dependent and thus they must create a Process Control File. Also, the CERES subsystems are implemented using certain data sampling and production strategies. The CERES subsystems use environment variables to define sampling and production strategies which in turn are used to name input and output files. Here, we satisfy those requirements for the SARB Main processor (CER5.1P1):

- `cd $CERESHOME/sarb/bin/sarb`
- `source ssit-main-env-Terra.csh62`
- `rm_script_5.1P1 CER5.1P1_PCF_$INSTANCE63`
- `ascii_gen_5.1P1 $DATE64`
- `pcf_gen_5.1P1 CER5.1P1_PCFin_$INSTANCE65`
- `runsarb CER5.1P1_PCF_$INSTANCE Main > Mac1_1118b.txt66`

We are expecting the program to take almost two hours of wall clock time but it terminates after only a few seconds! Since I redirected the standard output to the file, Mac1\_1118b.txt, I list out the file to find:

#### Segmentation fault (core dumped)

Is that all it's going to tell me? That was my first thought the first time I saw that message. Recall, that in section 3.4 The Console Utility, I described the Console utility and the CrashReporter logs. For this example, I have screen captured the CrashReporter log and other application windows that I used to track down the segmentation fault. Hopefully, you will be able to use this example to help solve similar problems that you will be encountering. After I collected my wits and recovered from the initial shock of my test case having failed so quickly, I opened the Console Utility (see Figure 12 Console Utility Window, Figure 13 Log Selection Controls, and Figure 14 InstSARB\_Drv.exe.crash.log Window). The CrashReporter log for our example is illustrated in Listing 6.

```
.....
Host Name:      cts1-105.larc.nasa.gov
```

<sup>62</sup> We source this C-shell script to define the test case parameters including the sampling strategy, production strategy, configuration codes, and the date for the test case. We will be running a case for 1 hour of data.

<sup>63</sup> In the previous step the environment variables, DATE and INSTANCE were defined based on the specified date, sampling strategy, production strategy, configuration codes and so on. This script removes any output files from previous runs.

<sup>64</sup> This is the basis for creating the Process Control File (PCF) as required by the PGS Toolkit. For SARB, this script creates an ASCII file that is used as an input file to the script that creates the PCF.

<sup>65</sup> This script creates the PCF required for the SARB Main processor for this particular hour of data.

<sup>66</sup> This script executes the SARB Main for a given test case. The SARB Main has a post-processor that runs automatically unless you eliminate it by specifying "Main" as the second argument on the command line. I have modified the SARB Main source to print diagnostic data to the standard output so I have redirected standard out to the file named Mac1\_1118b.txt.

## CERES Conversion Guide

```

Date/Time:      2004-11-18 15:18:07 -0500
OS Version:     10.3.5 (Build 7M34)
Report Version: 2

Command: InstSARB_Drv.exe
Path:          /CERES/sarb/bin/sarb/InstSARB_Drv.exe
Version:      ??? (???)
PID:          28482
Thread:       0

Exception:      EXC_BAD_ACCESS (0x0001)
Codes:         KERN_INVALID_ADDRESS (0x0001) at 0xfd5b89c0

Thread 0 Crashed:
0  InstSARB_Drv.exe  0x00034904  __with_cloud_MOD_cldlyr_id_ + 0x1e4
1  InstSARB_Drv.exe  0x00035020  __with_cloud_MOD_withcloud_profile_build_ + 0x6c0
2  InstSARB_Drv.exe  0x000357d4  __with_cloud_MOD_cloudpressure_ + 0x274
3  InstSARB_Drv.exe  0x000111f4  __fl_setup_MOD_inittune_drv_ + 0x114
4  InstSARB_Drv.exe  0x00005a64  __foot_drv_MOD_foot_proc_ + 0x264
5  InstSARB_Drv.exe  0x0000a988  main + 0x108
6  InstSARB_Drv.exe  0x00001c68  _start + 0x188 (crt.c:267)
7  dyld             0x8fela558  _dyld_start + 0x64

PPC Thread State:
srr0: 0x00034904 srr1: 0x0200f930          vrsave: 0x00000000
cr: 0x4840844a  xer: 0x00000000  lr: 0x00034748  ctr: 0x00000000
r0: 0xbfff5f80  r1: 0xbfff5f20  r2: 0x00034748  r3: 0xbfff6194
r4: 0xbfff619c  r5: 0xbfff5f84  r6: 0xbfff60a0  r7: 0xbfff5f80
r8: 0x005b89cc  r9: 0x00000002  r10: 0x005b8fc0  r11: 0xfcfffff4
r12: 0x0000009c  r13: 0xbfff5f84  r14: 0x00000004  r15: 0x00000011
r16: 0xbfff5f20  r17: 0x0000000e  r18: 0x0000000d  r19: 0x00000004
r20: 0x00000004  r21: 0x00000004  r22: 0x00000004  r23: 0x00000090
r24: 0x005b8284  r25: 0x0025f850  r26: 0x00000005  r27: 0x00000003
r28: 0x0000000c  r29: 0x00000008  r30: 0x7fffffff  r31: 0x7fbfffff

Binary Images Description:
0x1000 - 0x25dfff InstSARB_Drv.exe /CERES/sarb/bin/sarb/InstSARB_Drv.exe
0x8fe00000 - 0x8fe4ffff dyld /usr/lib/dyld
0x90000000 - 0x90122fff libSystem.B.dylib /usr/lib/libSystem.B.dylib
0x93a50000 - 0x93a54fff libmathCommon.A.dylib /usr/lib/system/libmathCommon.A.dylib
0x94640000 - 0x94649fff libz.1.dylib /usr/lib/libz.1.dylib
0xd4105000 - 0xd45defff libxlf90.A.dylib /opt/ibmcomp/lib/libxlf90.A.dylib
0xd4905000 - 0xd4906fff libxlfmath.A.dylib /opt/ibmcomp/lib/libxlfmath.A.dylib

```

### Listing 6 – CrashReporter Log

The entry, “Exception: EXC\_BAD\_ACCESS (0x0001)” confirms that there is a memory access problem as is implied by the original segmentation fault message. Huh? We are not trying to trap array boundary violations; rather we are trying to trap floating point problems. An interesting problem, indeed! Looking at the top entry in the call stack we have,

```
0  InstSARB_Drv.exe  0x00034904  __with_cloud_MOD_cldlyr_id_ + 0x1e4
```

At a glance I can see that the problem occurs in Module With\_Cloud.f90 in subroutine CldLyr\_ID. This module is in SARBlib<sup>67</sup>. I’m missing only one more piece of information; the line number in subroutine CldLyr\_ID where the problem occurs. When we re-built SARBlib in preparation for this test, we asked for an object and source listing. We can use the object listing to find the source line number. The top entry in the call stack has the relative address (0x1e4) of the source line that caused our problem. Recall that the module listing files will have the same name as the source module except for the suffix which will be .lst. Using Xcode, I opened the listing file, With\_Cloud.lst, and scrolled down to the object listing for subroutine CldLyr\_ID as is illustrated in Figure 16.

<sup>67</sup> Since SARBlib does not include a test suite, we get to test it “on the fly” when we are trying to test the other PGEs.



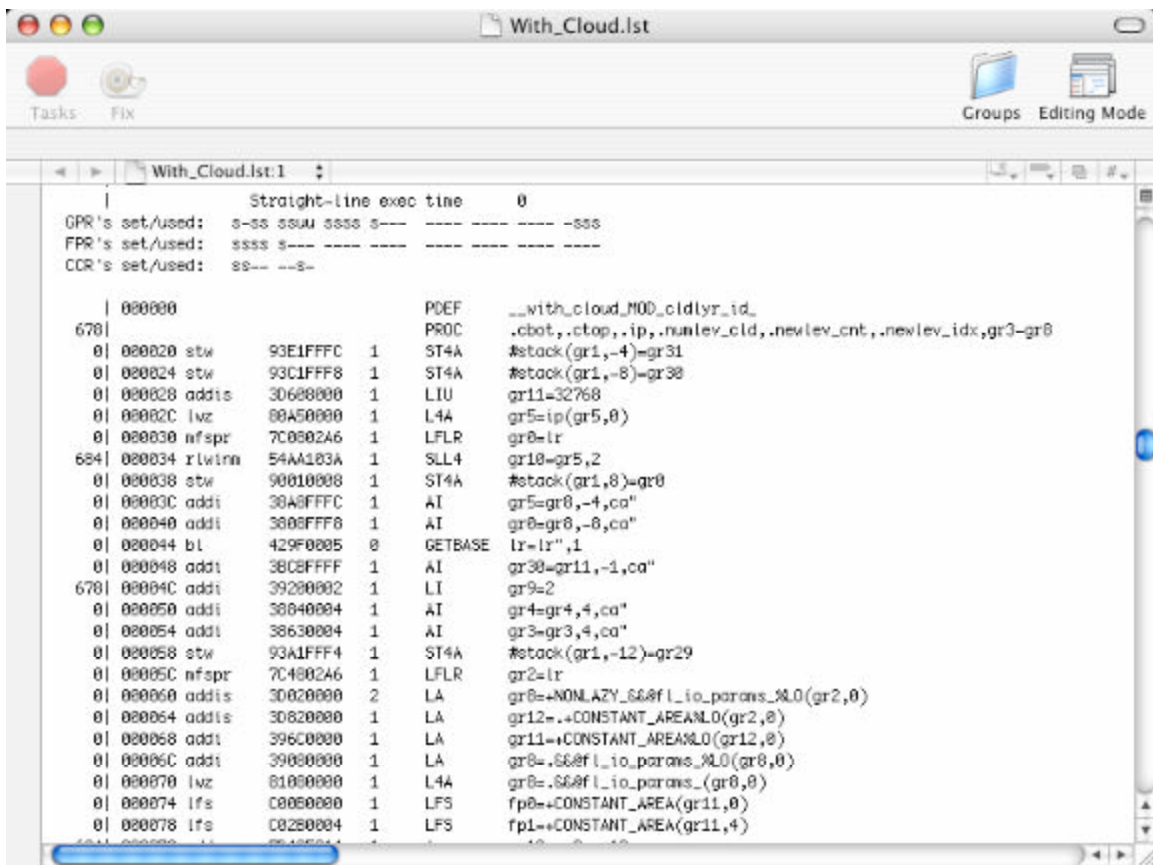


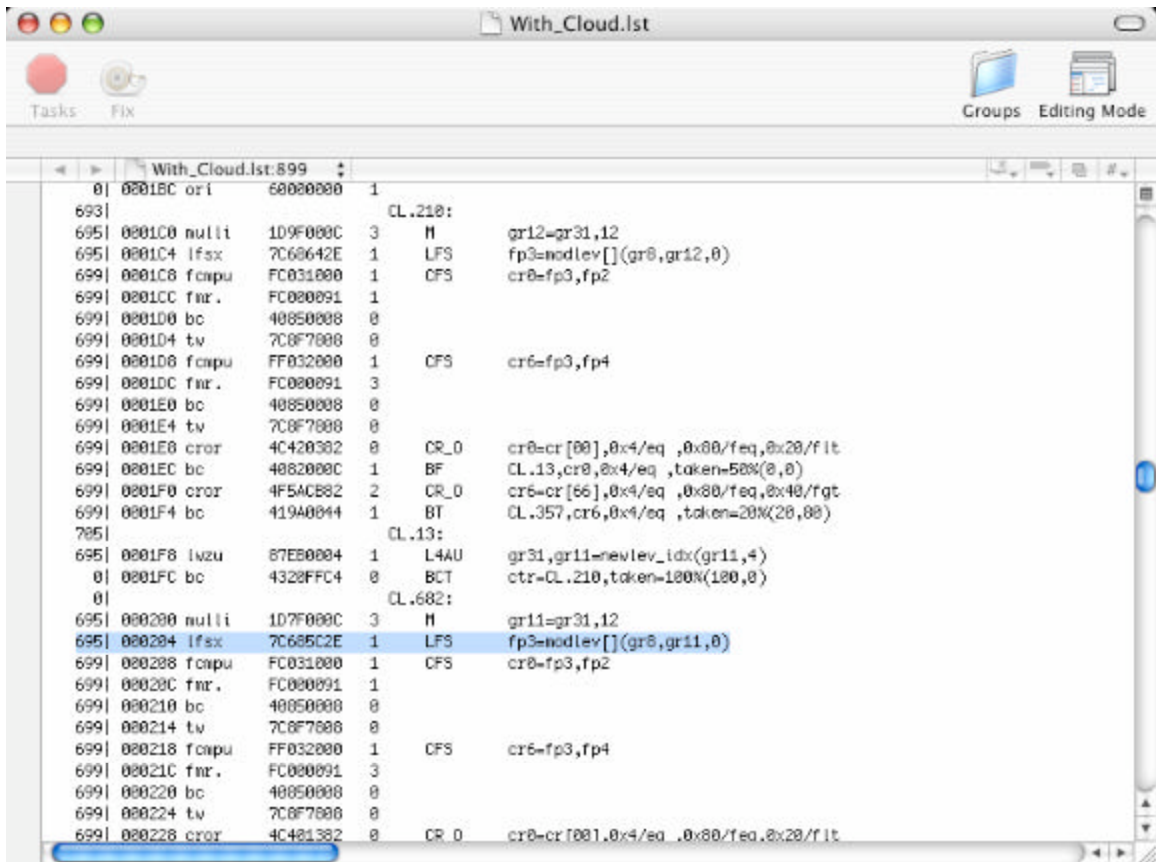
Figure 16 – Object Listing Header for Subroutine CldLyr\_ID

The header line is indicated by the PDEF mnemonic. The source line numbers are indicated in the left-most column. I can see that the PROC line (Subroutine CldLyr\_ID) occurs in source line 678. To the right of the source line numbers following the ‘I’ symbol we have the machine code instruction address column. Note that the PDEF entry occurs at address 000000 and the first ST4A entry is at address 000020. The first STA4 entry is the first executable address in the subroutine, and we note that it occurs at hexadecimal address 000020. Using our calculator we add the address from the top of the call stack to the 0x020 value. We have:

$$0x1e4 + 0x020 = 0x204$$

Now, we want to scroll down the relative address column until we find the computed address, 0x204. Figure 17 illustrates the object line highlighted at the location of the fault.

## CERES Conversion Guide



**Figure 17 – Object Address of Fault**

I highlighted the object line that has the relative address, 0x204. To the left, of the object address we can see the source line number is 695. Now we can look at the source listing at line 695 in module, With\_Cloud.f90 to see if we can tell what happened. I used Xcode to open module With\_Cloud.f90, and Figure 18 illustrates the mechanism for going directly to line 695. When you use the Xcode “Goto” dialogue it highlights the targeted line. The line at 695 is an array reference but if you look at it again you will see that we are using the term, “NewLev\_Idx(K)” as the column reference to array, ModLev. Since we had a segmentation fault, we can guess that either IP or NewLev\_Idx(K), or both, are outside the declared array boundaries for ModLev. Further scrutiny of the source code reveals that both IP and NewLev\_Idx are passed to subroutine CldLyr\_ID as arguments.

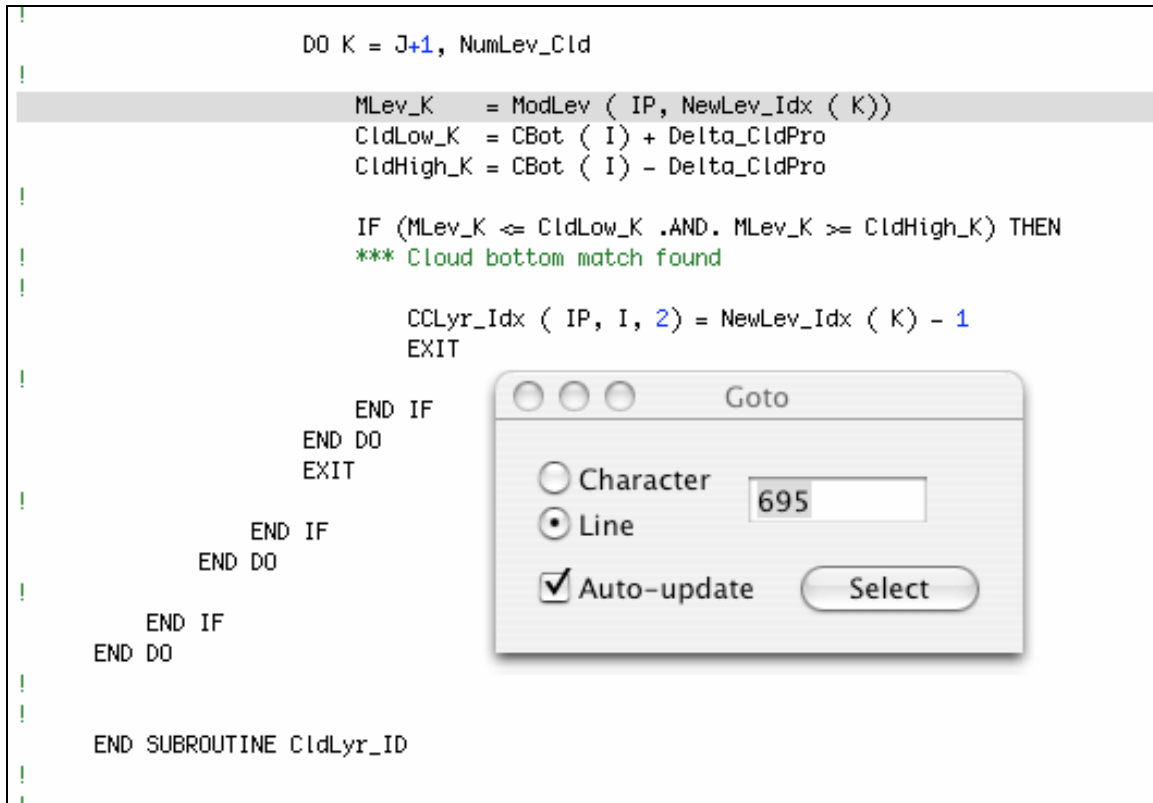


Figure 18 – Using Xcode to Find and View Source Line

Inserting print diagnostics revealed that IP was OK and NewLev\_Idx(4) was a huge number.

```

[CTS1-105:sarb/bin/sarb] Donaldsn% more Mac1_1118b.txt
NewLev_Idx uninitialized = 2143289343 2143289343 2143289343 2143289343
NewLev_Idx uninitialized = 2143289343 2143289343 2143289343 2143289343
NewLev_Idx uninitialized = 2143289343 2143289343 2143289343 2143289343
NewLev_Idx uninitialized = 2143289343 2143289343 2143289343 2143289343
NewLev_Idx uninitialized = 2143289343 2143289343 2143289343 2143289343
NewLev_Idx uninitialized = 2143289343 2143289343 2143289343 2143289343
NewLev_Idx uninitialized = 2143289343 2143289343 2143289343 2143289343
SR CldLyr_ID:
FP 4 : NewLev_Idx( 4 ) is out of range with value = 2143289343
NewLev_Idx(1:4) = 8 12 13 2143289343
Segmentation fault (core dumped)

```

Listing 7 – Terminal Window Diagnostic Output

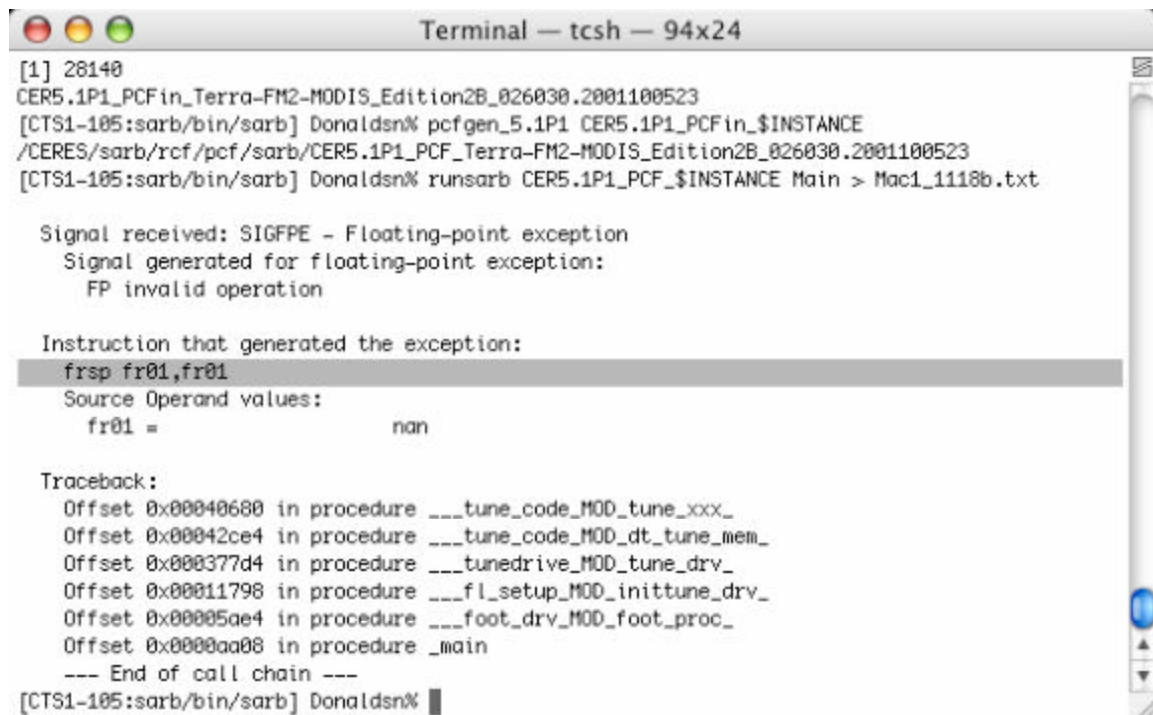
The whole diagnostic output is illustrated Listing 7. Grab your calculator and convert 2143289343 to hexadecimal to get 0x7fbfffff. Remember that number? We used it to initialize all local (AUTOMATICs) variables as specified in our compiler switch configuration. So we have defined the problem. Array NewLev\_Idx is a partially undefined local variable that is created in the call chain above subroutine CldLyr\_ID. I passed this problem on to the SARB development team for resolution.

But where was the invalid floating point operation? There was no floating point problem here but this example serves to illustrate the importance of initializing local variables. If we had used the `-qsave` compiler switch, we would have never seen the problem because `-qsave` causes local variables to be treated as STATIC variables. `-qinitauto` only initializes local or AUTOMATIC variables. So, if we had used `-qsave`, array

NewLev\_Idx would have been filled with zeroes each time it was created at runtime. Even though a zero value is outside of the array bounds for ModLev, it probably would not have caused a segmentation fault because the index address would be a reasonable address as opposed to something like 2,143,289,343. Our next example will actually find an invalid floating point operation.

#### 4.5 Tracking Down an Invalid Floating Point Operation

The preparation for this example is the same as the last example so we can skip right to the good part where we track down the invalid floating point operation. This example occurred prior to the last example during test operations on the SARB Main processor. Figure 19 Illustrates the terminal window just after the problem occurred:



```

Terminal — tcsh — 94x24

[1] 28140
CER5.1P1_PCFin_Terra-FM2-MODIS_Edition2B_026030.2001100523
[CTS1-105:sarb/bin/sarb] Donaldsn% pcfgen_5.1P1 CER5.1P1_PCFin_$INSTANCE
/CERES/sarb/rcf/pcf/sarb/CER5.1P1_PCF_Terra-FM2-MODIS_Edition2B_026030.2001100523
[CTS1-105:sarb/bin/sarb] Donaldsn% runsarb CER5.1P1_PCF_$INSTANCE Main > Mac1_1118b.txt

Signal received: SIGFPE - Floating-point exception
Signal generated for floating-point exception:
FP invalid operation

Instruction that generated the exception:
frsp fr01,fr01
Source Operand values:
fr01 = nan

Traceback:
Offset 0x00040680 in procedure __tune_code_MOD_tune_xxx_
Offset 0x00042ce4 in procedure __tune_code_MOD_dt_tune_mem_
Offset 0x000377d4 in procedure __tunedrive_MOD_tune_drv_
Offset 0x00011798 in procedure __fl_setup_MOD_inittune_drv_
Offset 0x00005ae4 in procedure __foot_drv_MOD_foot_proc_
Offset 0x0000aa08 in procedure _main
--- End of call chain ---
[CTS1-105:sarb/bin/sarb] Donaldsn%

```

Figure 19 - Terminal Window Traceback

Reading the information in the terminal window tells us that we have a Floating-point Exception, the instruction was “frsp fr01, fr01”, the source operand, fr01, contains a NaN<sup>68</sup>, and the problem occurred in module Tune\_Code.f90 in Subroutine tune\_xxx. But, notice that we don’t have any hexadecimal address information that would facilitate the determination of the source line in tune\_xxx. We resolve this problem by opening up the Console utility and consulting the CrashReporter logs. The CrashReporter has what we need:

```

*****
Host Name:      cts1-105.larc.nasa.gov
Date/Time:     2004-11-18 13:17:35 -0500
OS Version:    10.3.5 (Build 7M34)
Report Version: 2

Command: InstSARB_Drv.exe

```

<sup>68</sup> Not a Number!

## CERES Conversion Guide

```

Path:      /CERES/sarb/bin/sarb/InstSARB_Drv.exe
Version:   ??? (???)
PID:       28277
Thread:    0

Exception:  EXC_ARITHMETIC (0x0003)
Code[0]:   0x00000003
Code[1]:   0xeb004090

Thread 0 Crashed:
0  InstSARB_Drv.exe  0x00040680  __tune_code_MOD_tune_xxx_ + 0x1200
1  <<00000000>>      0x44000000  0 + 0x44000000
2  InstSARB_Drv.exe  0x00042ce4  __tune_code_MOD_dt_tune_mem_ + 0x1184
3  InstSARB_Drv.exe  0x000377d4  __tunedrive_MOD_tune_drv_ + 0x214
4  InstSARB_Drv.exe  0x00011798  __fl_setup_MOD_inittune_drv_ + 0x638
5  InstSARB_Drv.exe  0x00005ae4  __foot_drv_MOD_foot_proc_ + 0x264
6  InstSARB_Drv.exe  0x0000aa08  main + 0x108
7  InstSARB_Drv.exe  0x00001ce8  _start + 0x188 (crt.c:267)
8  dyld              0x8fela558  _dyld_start + 0x64

PPC Thread State:
srr0: 0x00040680 srr1: 0x0210f930      vrsave: 0x00000000
cr: 0x4e888444 xer: 0x20000000 lr: 0x000400d0 ctr: 0x00000000
r0: 0x44000000 r1: 0xbfff4330 r2: 0x00000003 r3: 0x00000007
r4: 0xbfff4348 r5: 0x00000006 r6: 0x0023f9f0 r7: 0x00000006
r8: 0x0023f9f0 r9: 0xbfff4638 r10: 0xbfff4828 r11: 0x00000005
r12: 0xbfff5b94 r13: 0xbfff6168 r14: 0xbfff5bfb r15: 0x00000054
r16: 0x00000054 r17: 0xbfff5ba0 r18: 0x00000060 r19: 0x0062d40c
r20: 0x00000014 r21: 0xbfff4d0c r22: 0x0000000c r23: 0x00000000
r24: 0x00000000 r25: 0x005bc4f0 r26: 0xbfff4638 r27: 0xbfff53a0
r28: 0xbfff4384 r29: 0x0062d410 r30: 0xbfff5e34 r31: 0x0003f4a8

Binary Images Description:
0x1000 - 0x25dfff InstSARB_Drv.exe /CERES/sarb/bin/sarb/InstSARB_Drv.exe
0x8fe00000 - 0x8fe4ffff dyld /usr/lib/dyld
0x90000000 - 0x90122fff libSystem.B.dylib /usr/lib/libSystem.B.dylib
0x93a50000 - 0x93a54fff libmathCommon.A.dylib /usr/lib/system/libmathCommon.A.dylib
0x94640000 - 0x94649fff libz.1.dylib /usr/lib/libz.1.dylib
0xd4105000 - 0xd45defff libxlf90.A.dylib /opt/ibmcomp/lib/libxlf90.A.dylib
0xd4905000 - 0xd4906fff libxlfmath.A.dylib /opt/ibmcomp/lib/libxlfmath.A.dylib

```

Listing 8 - CrashReporter Log for SR tune\_xxx

According to the CrashReporter, the invalid floating point operation occurred in Subroutine tune\_xxx at relative address 0x1200. We refer to the Tune\_Code.lst file and scroll down until we find the object header for tune\_xxx. Figure 20 illustrates the header line and as before, you can see the PDEF line is at relative address 000000.

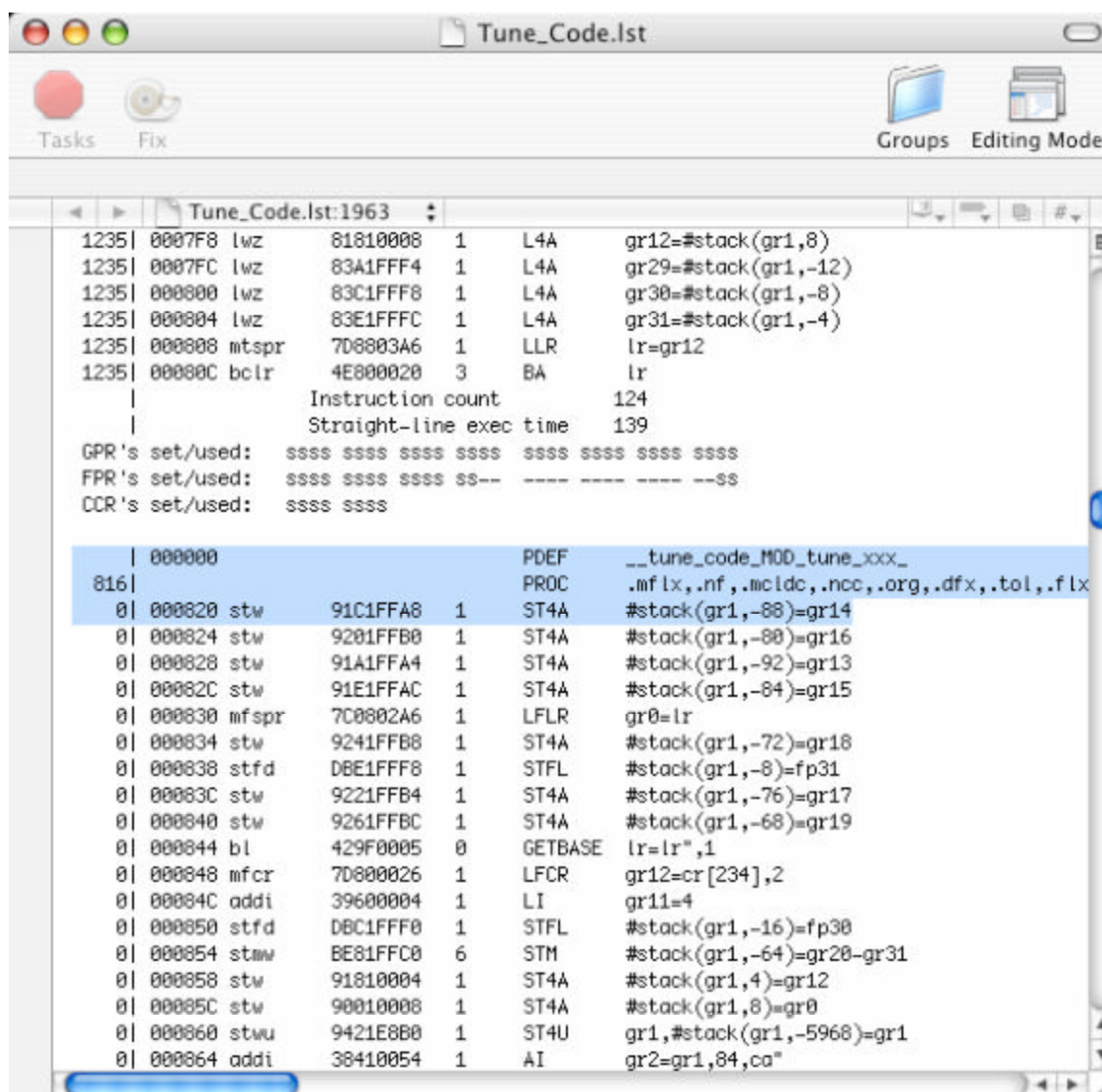


Figure 20 - Object Header Line for Subroutine tune\_xxx

The first executable machine instruction occurs at relative address 000820. We add the relative address from the CrashReporter log to the address of the first executable instruction in Subroutine tune\_xxx and we have:

$$0x1200 + 0x0820 = 0x1A20$$

We scroll down the relative address column until we find the computed relative address and we have:



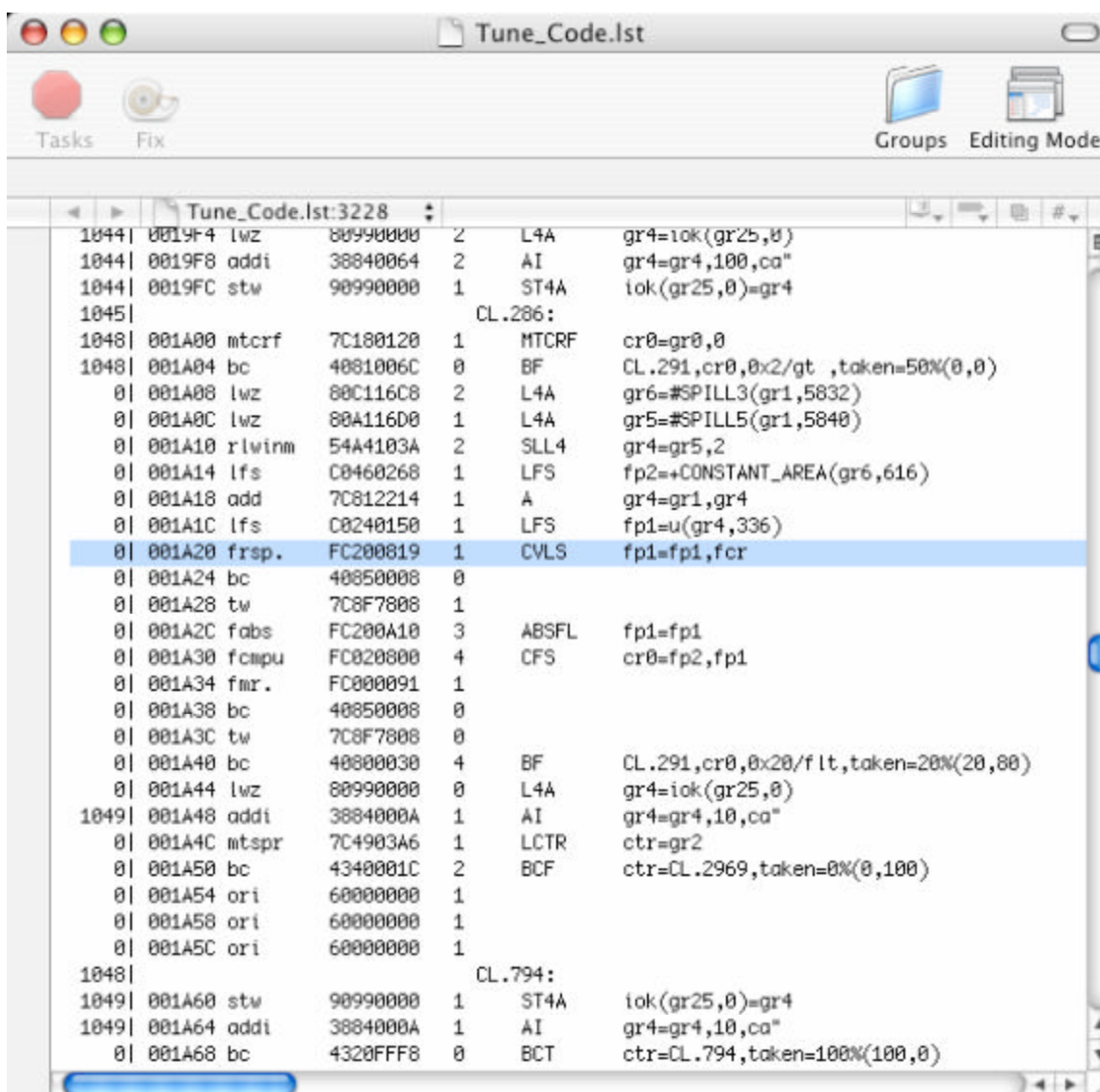
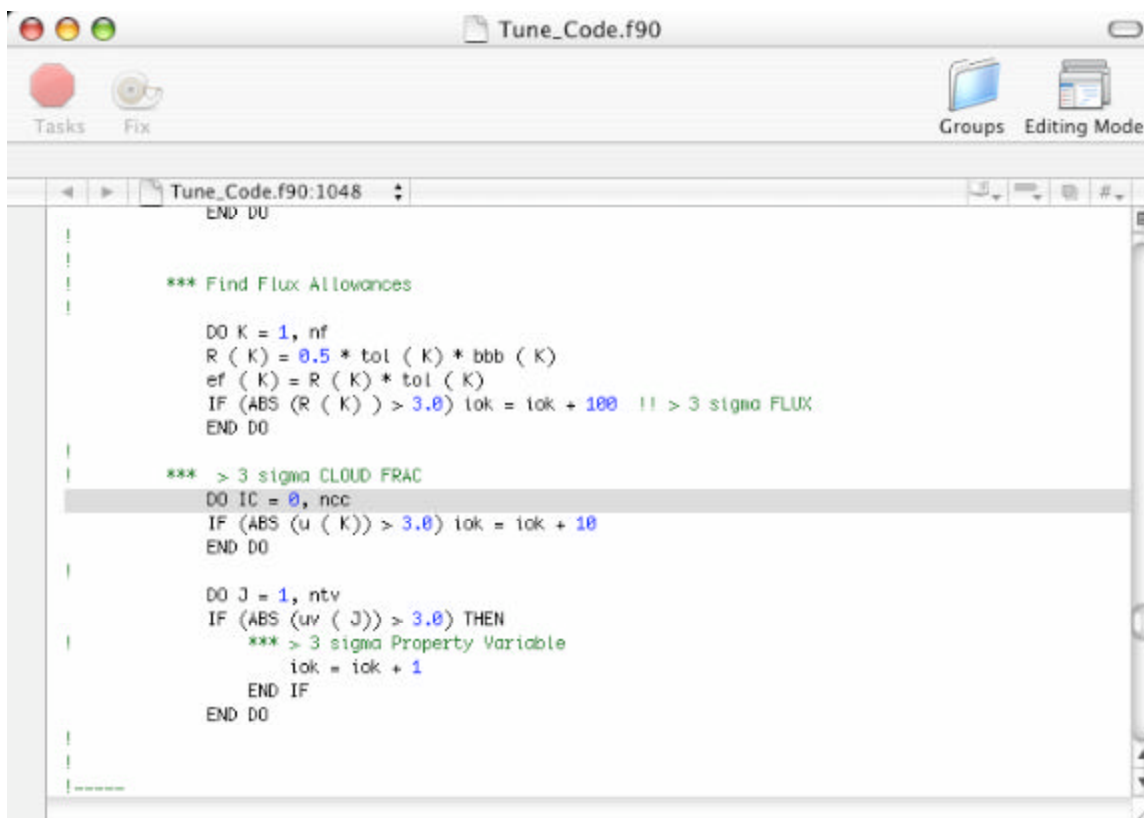


Figure 21 - Object Address of Invalid Floating Point Operation

When we look to the left-most column to identify the source line number we see a zero. In this case we trace up the column until we find a non-zero source line number, and in this example we find source line number 1048. Notice that the machine instruction at relative address 001A20 is an “frsp” instruction as was indicated by the terminal window trace back report. We are in the right place. We now use Xcode to open the source file and find line 1048. For module Tune\_Code.f90, we find:



**Figure 22 - Offending Source Line in Tune\_Code.f90**

Figure 22 reveals that source line 1048 does not contain a floating point operation. We must be on the wrong line. Remember that our code is also subject to optimization by the compiler and we specified `-O2`. Also, note that there is only one line of code sandwiched between the DO header and the END DO statement<sup>69</sup>. The compiler optimization has more or less blurred the DO loop into a single line containing many machine instructions. Assuming that there really is an invalid floating point operation, it would have to be occurring in the expression:

`ABS (u ( K) ) > 3.0`

Remember that the traceback report in the terminal window stated there was a NaN used as a source operand. In the expression above, there are 2 source operands; 1) `u ( K)` and 2) `3.0`. Since 3.0 is a number, then we must conclude that `u(K)` is a NaN. When I solved this problem I looked at the array, `u`, to ascertain its declared array boundaries. Then, I started thinking about the range of the index variable, `K`. That's when I noticed that the DO loop index is the variable `IC`. So, `K` was not varying over a range, it was constant over the range of the loop. Huh? `K` was used in the DO loop just above the one we have been scrutinizing, and so `K` must have been whatever value it was when the previous loop terminated. Since there is no exit from the DO `K`-loop, then `K` was probably equal to `nf + 1`, and `nf` is greater than `ncc`, the upper bound of the DO loop in question. This is when I realized that `K` should be replaced with `IC`, and that this was a typographical error that showed up at runtime only if we were looking for it. So, how did

<sup>69</sup> I just can't resist. Did you notice that there is no indentation for the DO-`K` and DO-`IC` loops? I think it makes them hard to read. The DO-`J` loop is not indented either but the IF-THEN statement is indented giving a hint of an attempt at readability.



the NaN come into play? Well, `u` is a local variable, and because we specified `initauto=7fbffff`, `u` got initialized with signaling NaN's. Since `K` was outside the expected range of `u`, the DO IC-loop was feeding NaN's to the ABS intrinsic function. The program was set up to trap on references to signaling NaN's<sup>70</sup> and we got what we deserved.

## 4.6 Comparing Results with the Benchmark

Fortunately, The PPC970 hardware and the IBM XLF compiler are IEEE-compliant, as is the SGI source platform from which we are porting code. If one of the platforms had not been IEEE-compliant, then we would have been developing transition programs that take binary files from the source platform in order to translate them into a format that is readable by the target platform<sup>71</sup>.

We also have escaped the Big versus Little Endian problem since both platforms are Big Endian. The PowerPC hardware has built-in byte swapping capabilities but I am thankful that tinkering with such mechanisms is unnecessary. If we were working with an Endian mismatch, I think we would always be looking over our shoulder when numbers from the source platform did not agree with their counterparts from the target platform.

So, where does that leave us? My experience with the SARB conversion effort compels me to answer that we are mainly left with problems that could be avoided by employing good coding techniques. Before I started the SARB conversion I was warned by well-meaning folks that the code would be littered with statements like:

```
IF ( THETA .EQ. 0.1 ) THEN
```

The implication in this example is that no two dissimilar platforms will agree on absolute equality<sup>72</sup> for numbers that are difficult to approximate. Thus, it is possible that the source platform would take one logic path while the target would go happily down the opposite logic path, and consequently the two platforms would produce different results. Perhaps I missed some of these problems during the SARB conversion, but every time I saw code like this<sup>73</sup>, both platforms were in agreement. That does not mean you should populate your code with such expressions since I am sure you could conjure up an expression that would be treated differently by the source and target machines. This section addresses the several, avoidable coding practices that do lead to differences between two dissimilar but IEEE-compliant platforms. The avoidable practices include but are not limited to:

- Using mixed mode arithmetic, creating imprecise floating-point results
- Not protecting against divide by zero
- Allowing intrinsic functions to process illegal arguments
- Allowing array boundary violations<sup>74</sup>

<sup>70</sup> Remember that `0x7fbffff` is a single precision signaling NaN.

<sup>71</sup> For you old-timers, I did just that when I converted the old ERBE science code from the CDC Cyber platforms with their 60-bit words to SUN Sparc-2 platforms that at least claimed to be IEEE-compliant.

<sup>72</sup> In our example, the value, 0.1, has traditionally been a difficult number to approximate. Yup, floating point hardware only approximates real numbers!

<sup>73</sup> I saw a lot of code like this.

<sup>74</sup> IEEE-compliance can't help with this problem.

- Failure to initialize local variables
- Executing unnecessary code

The following subsections will address these avoidable practices since they are the culprits that I confronted during the SARB conversion but first a word from our sponsor.

## 4.6.1 Learning to Live with NaN's and INF's

If you are writing CERES production code, you should NOT be learning to live with NaN's and INF's<sup>75</sup>! To me, NaN's and INF's are like cockroaches; they like to stay in the dark and when you turn on the light, they scatter in all directions and disappear. The SARB development team would argue that they have 1) equipped their code to replace output records containing NaN's with default values, and 2) the number of records that have to be replaced because of NaN's is a fraction of a percent of the total. I believe this to be a true statement, and it may be valid for working with code on a single platform. But, living with NaN's and INF's becomes a problem when you attempt to port the software in question to a dissimilar platform. In the previous section, I listed several avoidable coding practices, and almost all of them lead to the generation of NaN's and/or INF's.

## 4.6.2 Mixed Mode Arithmetic



**Mac says 5 - If you need double precision accuracy, then every variable in the computation must originate with double precision value.**

Actually, mixed mode arithmetic is probably the worst of all the avoidable coding practices that cause differences in the output results between platforms. Mixed mode arithmetic does not directly cause the generation of NaN's or INF's but it leads to results that are imprecise, and even IEEE-compliant platforms are vulnerable to misleading results under these circumstances. Our example takes place in a subroutine far, far away, and the players are:

- ak is a double precision array passed in to the subroutine as an argument
- u0 is a single precision scalar passed in to the subroutine as an argument
- du0 is a single precision scalar, local variable
- epsilon is a double precision scalar declared as a local variable and initialized to a single precision constant value, 1.0E-15
- c2 is a double precision scalar declared as a local variable

At the top of the subroutine du0 and epsilon are assigned permanent values using the following expressions:

```
DATA epsilon / 1.0E-15 /
du0 = 1.0 / u0
```

<sup>75</sup> Not a Number (NaN) and INFINITE (INF)

In Table 3 we can observe that the Mac G5 value of epsilon is indeed a single precision value of 1.0E-15 but it is definitely not a double precision value of  $1 \times 10^{-15}$ . Table 3 reveals that the Mac G5 version of du0 compares very nicely with the SGI version, and this implies that Mac and SGI agree closely for u0 as well.

Most of the body of the subroutine in question is contained within a DO-loop (loop index is j) that traverses the range of array ak. For each trip through the loop, variable c2 is assigned a value based on the following logic:

```
c2 = ak(j) * ak(j) - du0 * du0
IF( ABS( c2 ) .LE. epsilon ) c2 = epsilon
```

Now things are starting to go foul because c2 is computed as the difference between a double precision result and a single precision result. Also, in the IF-conditional expression we are calling the single precision intrinsic function, ABS, with a double precision argument, and then we are comparing a single precision result with a double precision scalar. If the conditional expression evaluates to TRUE, c2 is assigned the value of epsilon whose value is suspicious because it originates from a single precision literal. This is a real anomaly that occurred on the 12<sup>th</sup> iteration through the main loop in the subroutine. On iteration 12, following the IF statement I recorded the values illustrated in Table 3:

Variable Name	Mac G5 Value	SGI Value
<i>du0</i>	1.108221173	1.10822117
<i>ak(12)</i>	1.10822117938346687	1.1082211977031404
<i>epsilon</i> <sup>76</sup>	0.100000000362749373E-14	0.100000000362749373E-14
<i>c2</i>	0.100000000362749373E-14	4.0604501094776424E-08

Table 3 - Mac G5 Versus SGI Mixed Mode Results

From Table 3 we can now make the following observations:

- The Mac G5 computed c2 as less than epsilon and the SGI did not
- The Mac G5 has du0 in agreement with ak(12) out to 9 places and the SGI has du0 in agreement with ak(12) out to 8 places<sup>77</sup>
- At single precision accuracy for both platforms du0 and ak(12) are equal numbers so the difference of their squares should ideally be zero
- There is garbage precision in the Mac G5 and SGI epsilon values

I computed my own set of values using my 20-year old calculator, and I got a 3<sup>rd</sup> set of numbers but they were in the same ball park as the SGI values. Does that mean the SGI is better than the Mac G5? I don't think so. Did we ask both platforms to do something ridiculous? I think so. I was able to get both machines back in agreement by promoting du0 to double precision and making the following modifications:

```
DATA epsilon / 1.0D-15 /
du0 = 1.0D0 / DBLE( u0 )
```

<sup>76</sup> It is with considerable surprise that I note that the SGI and the Mac G5 can create identically the same garbage! For anyone who cares, 1.0D-15 on SGI and the G5 is 0.100000000000000008E-14.

<sup>77</sup> This is interesting since single precision on the G5 and the SGI is accurate to 7 places, sometimes 8, but never 9.

Now epsilon is truly double precision. *Unfortunately, du0 is still in question because u0 only has its original single precision value.* Promoting u0 to double precision prior to the divide does not help us gain any precision. The other modifications are:

```
c2 = ak(j) * ak(j) - du0 * du0
IF( DABS( c2 ) .LE. epsilon ) c2 = epsilon
```

Following the modifications noted here we have the results illustrated in Table 4.

Variable Name	Mac G5 Value	SGI Value
du0	1.10822122841541204	1.108221228415412
ak(12)	1.10822117938346687	1.1082211977031404
epsilon	0.100000000000000008E-14	0.100000000000000008E-14
c2 <sup>78</sup>	-0.10867648270032202E-06	-0.680719818291208867E-07

Table 4 - Mac G5 Vs SGI Following Mods

The truth is that the SGI and the Mac G5 came to agreement on the anomaly described here because I was able to coerce the G5 into using more of the available garbage precision in du0 to compute a value greater than epsilon. If you need double precision accuracy, then every variable in the computation must originate with double precision value. In the context of this example, this means that u0 should have originated as a double precision value. It comes to our fair subroutine with only single precision and thus it limits our computation to single precision accuracy no matter how we declare our variables<sup>79</sup>.

### 4.6.3 Divide by Zero



**Mac says 6 – The lesson learned here is that you should work divide by zero protection into your code as you develop the code to avoid having to re-write your code when you try to port it to another platform.**

It never fails to amaze me when I look at code that uses unprotected division. I assure you, there is no double meaning here. Dividing by zero can create a NaN or an INF depending on how you do it. You might argue that the data the code is manipulating *should never be* zero. But it seems to me that the longer your code lives, the greater the chance that you will feed your code some zeroes to use as a divisor. This can happen by making modifications to subroutines that are “upstream” from a subroutine, and then by failing to update the downstream subroutine that actually commits the crime. Listing 9 is an example taken from an anonymous SARB subroutine.

<sup>78</sup> Note that both Mac and SGI have changed values for c2! The implication here is that the modifications need to be made on both platforms so we can treat the garbage precision with fairness (equality?).

<sup>79</sup> For the sake of completeness and fairness, both the SGI and Mac G5 get the same result from ABS and DABS using a double precision argument.

```

do j = 1, nq
  t0 = 0.0
  do i = 2, mdfs
    i1 = i - 1
    fx(i1,j) = exp ( - ( t(i1) - t0 ) / ug(j) )
    fy(i1) = expn(i1)
    xx = lamdan(i1) * ug(j)
    fz1(i1,j) = ( 1.0 - fx(i1,j) * fy(i1) ) / ( xx + 1.0 )
    fz2(i1,j) = ( fx(i1,j) - fy(i1) ) / ( xx - 1.0 )
    ub(i1,j) = ug(j) * beta(i1)

c----- 4/2/97 (7)
    if (ub(i1,j) .eq. 1.0) ub(i1,j) = 1.001
c----- 4/2/97 (7)

    fid(i,j) = fid(i1,j) * fx(i1,j) + fj(i1) * fz1(i1,j) +
1          fk(i1) * fz2(i1,j) +
1          fuq2(i1) / ( ub(i1,j) + 1.0 ) *
1          ( alfa(i) - alfa(i1) * fx(i1,j) )
    t0 = t(i1)
  enddo
enddo

```

Listing 9 – Code Excerpt for Divide by Zero Problem

Don't try to understand what this code does; that's not the point. I count four divide operators inside this double DO-loop. If I had developed this code, the first few questions I would need to answer are:

- Will ug(j) ever be zero?
- Will xx + 1.0 ever be zero?
- Will xx - 1.0 ever be zero?
- Will ub(i1, j) + 1.0 ever be zero?<sup>80</sup>
- Should I have accepted that job offer from Midnight Software Company?

With the experience of two SARB Main processor 1-hour test cases under my belt, I can safely say that xx will occasionally be equal to 1.0 for all practical single precision purposes. When xx does equal 1.0 there will be a divide by zero, fz2(i1, j) will be set to a NaN, and fid(i, j) will get all fluxed up since it is dependent on fz2. This will ruin a perfectly good footprint. To avoid the generation of NaN's, I alerted the SARB development team<sup>81</sup> and implemented the workaround fix illustrated in Listing 10.

<sup>80</sup> This code does provide for protection against using the expression (ub(i1, j) - 1.0). Also, it is not really fair to show this code out of context. If we could look back and see the upstream code, we might find that ug(j) has a very narrow range of values, and possibly the same would be true for beta(i1). Since ub is the product of ug and beta it might be possible to absolutely predict that ub will never be negative. If this is so, then there should be code-level documentation (comments) that affirms this.

<sup>81</sup> The SARB team assured me that this was no big deal; that flux values with NaN's would be filtered out and replaced with default values.

```

      Do j = 1, nq
      t0 = 0.0
      do i = 2, mdfs
      i1 = i - 1
      fx(i1,j) = exp ( - ( t(i1) - t0 ) / ug(j) )
      fy(i1) = expn(i1)
      xx = lamdan(i1) * ug(j)
c JLD Fix for xx == 1.0 causing divide by zero (fz2)
      if(xx == 1.0) then
      xx = 1.00001
      print*, ' qftisf: FP:', db_fp, ': fz2: xx = ', xx
      end if
      fz1(i1,j) = ( 1.0 - fx(i1,j) * fy(i1) ) / ( xx + 1.0 )
      fz2(i1,j) = ( fx(i1,j) - fy(i1) ) / ( xx - 1.0 )
      ub(i1,j) = ug(j) * beta(i1)

c----- 4/2/97 (7)
      if (ub(i1,j) .eq. 1.0) ub(i1,j) = 1.001
c----- 4/2/97 (7)

      fid(i,j) = fid(i1,j) * fx(i1,j) + fj(i1) * fz1(i1,j) +
1          fk(i1) * fz2(i1,j) +
1          fuq2(i1) / ( ub(i1,j) + 1.0 ) *
1          ( alfa(i) - alfa(i1) * fx(i1,j) )
      t0 = t(i1)
      enddo
      enddo

```

Listing 10 – Code Excerpt With Workaround for Divide by Zero

No, I did not prepare workaround fixes for all possible divide by zero cases; there were just too many. The lesson learned here is that you should work divide by zero protection into your code as you develop the code to avoid having to re-write your code when you try to port it to another platform.

Due to the precision problems noted earlier, the source platform and the target platform do not always agree on when and/or where a NaN or an INF will be created. Consequently, the target platform will produce output records that contain NaN's<sup>82</sup> when the source platform does not and vice versa.

<sup>82</sup> The SARB code filters out such NaN's and replaces them with that wonderful IEEE-compliant default number.

#### 4.6.4 Proper Usage of Intrinsic Functions



Mac says 7 - For single precision the argument to EXP must be less than or equal to 88.7228, and for double precision the argument to DEXP must be less than or equal to 709.7827.

I've already addressed using single precision intrinsic functions with double precision arguments. I think the two most abused intrinsic functions are LOG and EXP. Let's take a look at a code excerpt that routinely abuses both LOG and EXP.

```
IF (IPass == ConstrPass1) THEN                                ! Tuned
  Adjln_tau(CH_Idx) = &
    log(OptDepth_mn(IPass, CH_Idx)) - log(OptDepth_mn(InitPass, CH_Idx))

  OptDepthLin_mn(IPass, CH_Idx) = &
    EXP(log(OptDepthLin_mn(InitPass, CH_Idx)) + Adjln_tau(CH_Idx))

  fi%fc(CH_Idx)%tau_vis = OptDepth_mn(IPass, CH_Idx)
  fi%fc(CH_Idx)%sc%mn_lin_tau = OptDepthLin_mn(IPass, CH_Idx)
ENDIF
```

Listing 11 - Code Excerpt for LOG of Zero

In the first assignment statement following the IF-statement, array element OptDepth\_mn(InitPass, CH\_Idx) is frequently zero causing the intrinsic function LOG to generate an INF for the assigned value of array Adjln\_tau(CH\_Idx). Array Adjln\_tau(CH\_Idx) is then referenced in the second assignment statement following the IF-statement. This causes the intrinsic function EXP to be called with an INF. For single precision the argument to EXP must be less than or equal to 88.7228, and for double precision the argument to DEXP must be less than or equal to 709.7827.

To work around this problem, I applied the following modifications:

```
IF (IPass == ConstrPass1) THEN                                ! Tuned
! JLD Fix for avoiding taking the LOG of zero
  if(OptDepth_mn(IPass, CH_Idx) == 0.0 .or. OptDepth_mn(InitPass, CH_Idx) == 0.0) then
    Adjln_tau(CH_Idx) = Default_REAL4_FL
  else
    Adjln_tau(CH_Idx)=log(OptDepth_mn(IPass, CH_Idx)) - log(OptDepth_mn(InitPass, CH_Idx))
  end if
! JLD Fix for avoiding taking the LOG of zero or EXP of INF
  if(OptDepthLin_mn(InitPass,CH_Idx)==0.0 .or. Adjln_tau(CH_Idx)==Default_REAL4_FL) then
    OptDepthLin_mn(IPass, CH_Idx) = 0.0
  else
    OptDepthLin_mn(IPass,CH_Idx) = &
      EXP( log(OptDepthLin_mn(InitPass, CH_Idx)) + Adjln_tau(CH_Idx))
  end if

  fi%fc(CH_Idx)%tau_vis = OptDepth_mn ( IPass, CH_Idx)
  fi%fc(CH_Idx)%sc%mn_lin_tau = OptDepthLin_mn ( IPass, CH_Idx)
ENDIF
```

Listing 12 - Code Excerpt with Workaround for LOG of Zero

I was assured by the SARB development team that neither the original error nor my workaround code had any scientific impact. Also, I was unable to demonstrate any difference in the output files due to this particular anomaly.

## 4.6.5 Array Boundary Violations



**Mac says 8 - When an array bounds violation occurs during a store operation, then unpredictable damage can occur in another part of the program.**

Apparently the SGI platform is immune to accessing arrays outside of their declared bounds. This is not the case on the Mac G5 platform. On the G5 like the SGI, if you have not compiled your code to trap on array boundary violations, they will occur silently unless there is an attempt to write over the operating system kernel<sup>83</sup>. During the SARB conversion effort, I observed an uncorrected anomaly disappear following the correction of a problem where an array was being over-indexed by just one element. The following example occurred while testing the SARB Main processor with the compiler configured to generate code that will trap on any attempt to reference an array outside its bounds.

```

*****
Host Name:      cts1-97.larc.nasa.gov
Date/Time:      2004-10-05 14:17:08 -0400
OS Version:     10.3.5 (Build 7M34)
Report Version: 2

Command: InstSARB_Drv.exe
Path:    /CERES/sarba/bin/sarb/InstSARB_Drv.exe
Version: ??? (???)
PID:     4377
Thread:  0

Exception:  EXC_SOFTWARE (0x0005)
Code[0]:    0x00000001
Code[1]:    0x000214b8

Thread 0 Crashed:
0  InstSARB_Drv.exe  0x000214b8  __spectral_sfc_MOD_as_ocean_ + 0x198
1  <<00000000>>    0x00000018  0 + 0x18
2  InstSARB_Drv.exe  0x000244e8  __spectral_sfc_MOD_choose_spectral_properties_ + 0x3c8
3  InstSARB_Drv.exe  0x0000a7c8  __inst_sfcalb_MOD_sfcalb_drv_ + 0x88
4  InstSARB_Drv.exe  0x00008364  __instsarb_ingest_MOD_sfcrad_init_ + 0x124
5  InstSARB_Drv.exe  0x00009254  __instsarb_ingest_MOD_ingest_input_ + 0xb4
6  InstSARB_Drv.exe  0x00007294  __foot_drv_MOD_foot_proc_ + 0xb4
7  InstSARB_Drv.exe  0x0000ba90  main + 0x70
8  InstSARB_Drv.exe  0x00003448  _start + 0x188 (crt.c:267)
9  dyld             0x8fela558  _dyld_start + 0x64

```

**Listing 13 - CrashReporter Log for Array Bounds Violation**

I have truncated the traceback report because the call stack has all the information that we need for this problem. The relative address of the error (0x198) turned out to be source line number 1107 in module Spectral\_Sfc.f90 in Subroutine `as_ocean`. The offending line of code was:

```
IF ( tau < 2 ) angexp = AerConst_Angexp (DomAerType)
```

<sup>83</sup> I am not suggesting that the PGE should always run under this configuration. I am suggesting that the PGE be configured to trap array bounds violations for several different test cases to flush such problems out of the code before we send it to production.



Since I had compiled the source code to trap on array access violations and not on floating point operations, I can zero in on the variable, `DomAerType` because it is the only array index expression in the source line. As it turns out, `DomAerType` is defined in an upstream subroutine but its possible range of values is defined by the following declaration:

```
INTEGER, PARAMETER :: itrantyp_rgc(NumConst_MATCH) = (/ 24,25,18,1,11,10,9 /)
```

A look at the declaration for array `AerConst_Angexp` reveals that it has 18 elements in the range 1:18. In this particular case, `DomAerType` was 25, and thus it was clearly outside the bounds of `AerConst_Angexp`. A brief e-mail conversation with the SARB development team revealed that they had upgraded the code and they had forgotten to upgrade `AerConst_Angexp` to have 25 instead of 18 elements. The SARB team also assured me that this error had very little scientific significance.

Another mitigating factor in this example is that the array bounds violation occurs when the array is being referenced rather than stored. When an array bounds violation occurs during a store operation, then unpredictable damage can occur in another part of the program. In the case of an array reference that is outside the declared bounds, the source and target platforms will not necessarily agree on the value that gets referenced because the source and the target platforms handle storage allocation differently. There are various reasons for this including the existence of optimization techniques on both platforms that include padding the byte allocations of arrays to facilitate more efficient access. For this case, we are then clearly on our way to creating differing results on the respective platforms when we allow array bounds violations to occur silently.

### 4.6.6 Un-initialized Local Variables

In section 0, I documented an excellent example of a problem caused by an array that was declared as a local variable and it was never fully initialized. When I encountered that problem while testing SARB, I was amazed that it had remained undetected for so long, especially since I had deliberately screened for array bounds violations in prior tests<sup>84</sup>. When I checked my engineering notebook, I discovered that I had inadvertently left the `-qsave` compiler switch in the configuration setting for the compiler. The `-qsave` switch causes AUTOMATIC variables to become STATIC variables, and STATIC variables are not effected by the `-qinitauto=UglyNumber` compiler setting. At the time that I made this realization I asked myself, “to what value did the array elements get initialized?” An experiment on the SGI platform using a local, 4-element integer array revealed the values 0, 1, 0, 0. Executing the same code on the Mac G5 yields the values 0, 0, 0, 0. Although my experiment is not conclusive, it does prove that the two platforms do different things for undefined local variables. So, here is an avoidable situation that can cause different results between the two computer platforms.

---

<sup>84</sup> The problem showed up as a segmentation fault because I configured the compiler to initialize AUTOMATICs with the bit pattern, 0x7fbfffff. The undefined element of the local array was used as an array index that was beyond the memory capacity of the Mac G5.

### 4.6.7 Executing Unnecessary Code



**Mac says 9 - Multiplying a number that is very near the maximum size by another number greater than one will quite possibly cause an INFINITE result, and further operations on an INFINITE value have unpredictable results.**

I'm still reeling over this problem. A little background information might help to make this problem easier to understand. When I first began to familiarize myself with the SARB subsystem, I noticed the usage of so-called default values for real and integer variables. For example, SARB defines a value called `Default_REAL4_FL` for single precision real numbers. `Default_REAL4_FL` is slightly less than the IEEE maximum positive value for 32-bit floating point numbers. Further investigation led me to believe that the other CERES subsystems reference the same default settings and they write them to data files that are output by one subsystem and input to another. Later, I discovered that the SARB Main processor uses `Default_REAL4_FL` and other similar constants to set variables to unreasonable values in certain situations. A hypothetical, overly simplified example of this behavior could be something like:

- Variable ALBEDO is initialized to an unreasonable value, say `Default_REAL4_FL`
- The code is iteratively processing footprints (Fields of View)
- The current footprint was sampled in darkness (during the night part of the orbit)
- There is no albedo at night
- Call a subroutine that processes variable ALBEDO

If you are looking for something nasty here, you won't find it. My assumption would be that the subroutine that processes variable ALBEDO would perform a logic test like:

```
IF (ALBEDO .EQ. Default_REAL4_FL) THEN ...
```

One action that might be taken is to return to the caller. Another action might be to execute a different logic path that is cognizant that the data is from a night time footprint. Of course, the action I left for last is the one that treats variable ALBEDO as though the current footprint occurred in the day time. Well, so what. The assumption here is that the code will behave normally as it attempts to digest `Default_REAL4_FL` disguised as the variable, ALBEDO. Here is what happened to me when I was converting the SARB Main processor:

```

.....
Host Name:      cts1-97.larc.nasa.gov
Date/Time:     2004-10-12 11:51:12 -0400
OS Version:    10.3.5 (Build 7M34)
Report Version: 2

Command: InstSARB_Drv.exe
Path:         /CERES/sarba/bin/sarb/InstSARB_Drv.exe
Version: ??? (???)
PID:          7177
Thread:       0

Exception:     EXC_ARITHMETIC (0x0003)
Code[0]:       0x00000003
Code[1]:       0xf2809090

Thread 0 Crashed:
0  InstSARB_Drv.exe 0x0002c82c __flsa_lut_utils_MOD_rlui_ + 0xbcc
1  InstSARB_Drv.exe 0x00008090 __instsarb_ingest_MOD_toa_init_ + 0xb0
2  InstSARB_Drv.exe 0x0002d044 __flsa_lut_utils_MOD_flsasnow_lut_ + 0x1a4
3  InstSARB_Drv.exe 0x000250a8 __spectral_sfc_MOD_cldsnow_ + 0x168
4  InstSARB_Drv.exe 0x0002950c __spectral_sfc_MOD_choose_spectral_properties_ + 0xf8c
5  InstSARB_Drv.exe 0x0000a188 __inst_sfcalb_MOD_sfcalb_drv_ + 0x128
6  InstSARB_Drv.exe 0x00007908 __instsarb_ingest_MOD_sfcrad_init_ + 0x168
7  InstSARB_Drv.exe 0x00008ad4 __instsarb_ingest_MOD_ingest_input_ + 0x114
8  InstSARB_Drv.exe 0x00006588 __foot_drv_MOD_foot_proc_ + 0xc8
9  InstSARB_Drv.exe 0x0000b338 sarb_drv + 0x98
10 InstSARB_Drv.exe 0x00002968 _start + 0x188 (crt.c:267)
11 dyld             0x8fe1a558 _dyld_start + 0x64

```

Listing 14 – CrashReporter Log for Unnecessary Code Example

This problem was a little more difficult to trace since the function, `rlui` was an innocent victim<sup>85</sup>. Function `rlui` was passed a parameter with the value `Default_REAL4_FL`, and function `rlui`'s logic is designed to assume that the parameters that are passed in are in their proper ranges. So function `rlui` performed a multiplication operation with the parameter that was set to `Default_REAL4_FL`. Multiplying a number that is very near the maximum size by another number greater than one will quite possibly generate an INFINITE result, and further operations on an INFINITE value have unpredictable results. So `rlui` created an INF. When I passed this information on to the SARB development team they told me, "...this is another instance of calling a module even when it is not needed. Obviously Albedo does not exist at night..." They also suggested a fix that involved testing for `Default_REAL4_FL` in the upstream caller, function `flsasnow_lut` in this case. This problem occurred again because there turned out to be another caller, function `flsa_lut` that passed similar parameters to function `rlui`. Poor `rlui`! I'm not sure what bothers me more, `rlui` generating INF's or calling `rlui` during the night when we know `rlui` works the day shift.

So, what is my point? Recall that one of the objectives of the SARB conversion was to determine the feasibility of achieving 10x processing on the PPC970 platforms. The very notion of 10x processing implies that the CERES code should be as efficient and fast as possible to minimize the wall clock hours spent processing the hours of real-time data. If

<sup>85</sup> Well, maybe not. Would you write code that blindly accepted the values passed in as parameters? If you answered yes, I have some property in Florida that might interest you.

this is the case, then why would we tolerate the iterative execution of code that is unnecessary?

## 5 A Conversion Checklist

The checklist presented here should not be regarded as a “cookbook” approach to converting your subsystem; rather it should be regarded as a mechanism for determining readiness and progress.

- ? Configure a benchmark subsystem on the source platform
  - ? Establish at least one complete subsystem test case
  - ? Create a subsystem delivery package for source and data
  - ? Execute benchmark case and add expected outputs to delivery package
  - ? Verify/establish secure shell communications with target platform
- ? Acquire access to target platform
  - ? Verify presence of gcc environment
  - ? Verify presence of IBM XL FORTRAN compiler
  - ? Establish a development environment
  - ? Verify/establish secure shell communications with source machine
  - ? Verify/establish administrative permissions for */CERES/your\_subsystem*
  - ? Verify/establish PGS Toolkit residence on platform
  - ? Verify/establish CERES library residence on platform
- ? Develop/document plan for source to target conversion
- ? Develop/document schedule for source to target conversion
- ? Submit conversion plan and schedule to management for approval
- ? Implement conversion plan
  - ? Acquire subsystem delivery package(s) from source platform
  - ? Establish target platform directory structure
  - ? Install subsystem source files on target platform
  - ? Install at least one complete data case on target platform
  - ? Install expected output file comparison files on target platform
  - ? Convert scripts and make files to target platform environment
  - ? Install test suite software on target platform
  - ? Convert test suite software to target platform if necessary
  - ? Build each subsystem PGE and correct build errors
  - ? Execute each PGE and correct runtime errors
  - ? Implement/execute test suite software for each PGE
  - ? Establish comparison agreement with expected values for each PGE
  - ? Install and test another complete subsystem test case
  - ? Update conversion plan and schedule with deviations from plan
- ? Document results and lessons learned from the conversion effort

## 6 Findings from the SARB Conversion Effort

The SARB conversion effort is a special case in that it was the first attempt to quantify the effort involved with the migration of a typical CERES subsystem to a non-SGI platform. Since the SARB conversion was a first attempt, it also necessarily involved the migration of the PGS Toolkit and the CERES library, CERESlib. Fortunately, there was an existing version of the PGS Toolkit that had been ported to a Macintosh platform of unknown ilk. The following subsections will summarize the experience and outcome from converting the PGS Toolkit, CERESlib, and the SARB subsystem to the PPC-970 platform<sup>86</sup>.

### 6.1 The PGS Toolkit Installation on the Mac G5

My engineering notebook has me downloading the Macintosh Darwin version of the PGS Toolkit installation files (hereafter referred to as TK) on April 12, 2004. The TK software encompasses several components and so the TK installation is broken into several software package installations. Each time I installed the TK on the Mac G5, I updated the SARB Conversion Plan with corrections to the install steps with the result that the SARB Conversion Plan is currently an excellent description of the process at the lowest level. The Mac-Darwin version that I downloaded and installed is version 5.2.10<sup>87</sup> which incorporates zlib version 1.1.4, JPEG version 6b, HDF4 version 4.2r0, HDF5 version 1.6.1, HDF-EOS version 2, and HDF-EOS version 5. The following subsections will detail the specific TK installation findings in the order of the component installations.

#### 6.1.1 Installing zlib

The zlib installation is a standalone installation that incorporates the ‘configure’ utility to build a make file after analyzing the target platform. The resultant make file will fail because it assumes that the target platform does not already have a version of zlib installed. Using a text editor, open the resultant file named “Makefile” and change the LDFLAGS entry from “LDFLAGS=-L. -lz” to “LDFLAGS=libz.a”. The zlib installation will then proceed without any further problems. I installed zlib at /usr/local/zlib-1.1.4/lib/libz.a and /usr/local/zlib-1.1.4/include.

#### 6.1.2 Installing JPEG

The JPEG install also incorporates the ‘configure’ utility to build a make file after analyzing the target platform. The resultant make file has several functions including the build, a test, an install, a library install, and a header file install. All goes well until the so-called “make install” step which fails due to the fact that the target subdirectories do not exist prior to the installation. I dealt with this problem by iteratively allowing the “make install” step to fail to identify the non-existent subdirectories. I created each

---

<sup>86</sup> Unfortunately, the findings are based on experiences with just the Macintosh G5 rather than the G5 and the IBM PPC-970 cluster nodes yet to be declared ready by the DAAC.

<sup>87</sup> I understand that there is now a more recent version of the PGS Toolkit installation that corrects many of the errors and shortcomings of the original Mac-Darwin set of installation files.

subdirectory and re-started the “make install” step until it completed normally. I installed JPEG at `/usr/local/jpeg-6b`<sup>88</sup>.

### 6.1.3 Installing HDF4

The HDF4 software requires the previous zlib and JPEG installations to be complete prior to the installation. Like the zlib and JPEG installations, the HDF4 installation incorporates the ‘configure’ utility to create a make file after analyzing the target platform. Unfortunately, this version of the installation assumes that the Mac-Darwin platform is using the g77 FORTRAN compiler, and of course, that is not the case. To resolve this problem, we locate the “powerpc-apple” configuration file in the “config” subdirectory<sup>89</sup>, and edit the configuration file to conform to the IBM XLF compiler switch settings that are equivalent to the g77 settings<sup>90</sup>. Following the compiler configuration modifications, the HDF4 installation proceeded smoothly. I installed HDF4.2r0 in `/opt/net/TOOLKIT/hdf/macintosh/HDF4.2r0`.

### 6.1.4 Installing HDF5

The README file that accompanies the Mac-Darwin distribution of the TK recommends that HDF5 be installed using the TK installation script. Based on advice that I received from the SGI system administrator, I installed HDF5 as a standalone installation in a similar manner to the HDF4 install. See the SARB Conversion Plan for details but the HDF5 install and test proceeded without difficulty.

### 6.1.5 Installing HDF-EOS Version 2

Contrary to the README file in the installation kit, I installed HDF-EOS version 2 as a standalone build. Unfortunately, the Mac-Darwin distribution of HDF-EOS2 contains an error that must be corrected to avoid problems that will surface after the TK installation has been completed. The error is that the wrong version of libGctp.a was distributed as a pre-built library. The HDF-EOS2 distribution kit does provide the source files for libGctp.a and I was able to build the correct version of libGctp.a<sup>91</sup>. This was not an easy fix, so see the SARB Conversion Plan for a step by step process. HDF-EOS2 is dependent on HDF4, so the HDF4 installation must have been completed prior to the HDF-EOS2 install.

I contacted the PGS Toolkit maintenance team about this problem and they confirmed the error. They suggested two alternatives; 1) they provided a make file to build libGctp.a from the existing source files but they did not provide any instructions on how to perform the correction in the context of the installation process, and 2) they suggested scrapping the distribution that I had in favor of their most recent distribution. I went with my own fix but the 2<sup>nd</sup> suggestion was tempting since it appeared to be a generic distribution that apparently now includes the Mac-Darwin platform as a normal maintained platform

---

<sup>88</sup> See the SARB Conversion Plan for the details.

<sup>89</sup> When the installation files are extracted from the delivery tar file, a basic subdirectory structure is established and referenced by the various install scripts.

<sup>90</sup> It was not that simple; see the SARB Conversion Plan for all the details.

<sup>91</sup> The HDF-EOS2 install will proceed without error and unless you incorporate a specific test for libGctp.a, there will be no indication that there is a problem until you attempt to install CERESlib and find that there are many unsatisfied externals when you try to build the library. At this point it is not obvious where the missing externals should come from so there is no trail of bread crumbs to follow.

rather than a special case. Future conversions to the Mac G5 may want to consider upgrading to the most recent distribution release.

### 6.1.6 Installing HDF-EOS Version 5

Contrary to the README file in the installation kit, I installed HDF-EOS version 5 as a standalone build. The installation process as documented in the SARB Conversion Plan went smoothly. However, HDF-EOS5 uses the same library file (libGctp.a) as HDF-EOS2, and HDF-EOS5 also includes libGctp.a as a pre-built library in the distribution files. The SARB Conversion Plan includes a step that corrects this problem by copying the HDF-EOS2 copy of libGctp.a to the appropriate HDF-EOS5 subdirectory.

### 6.1.7 Installing the Toolkit

The TK part of the installation was not without stress either. I have documented the complete process in the SARB Conversion Plan so refer to it for the exact details as I am only summarizing the event here.

#### 6.1.7.1 IBM XLF and Not g77

The C-shell script, INSTALL-Toolkit, had to be modified to correct the assumption that the g77 FORTRAN compiler is in use on the target Mac G5. The g77 compiler flags were updated to their equivalents for the IBM XLF compiler.

#### 6.1.7.2 Searching for search.h

The TK distribution for the Mac-Darwin special case was apparently prepared on a deficient Macintosh platform. There is a note in the README file that comes with the distribution that states that the `search.h` header file was not present on the Macintosh platform that was employed to create the distribution files. A substitute was provided but there was a shortcoming with the substitute file as well. Header file `search.h` is included in the gcc standard C distribution for the Macintosh G5 platform, so I removed the substitute version and the problem was corrected.

#### 6.1.7.3 I Can't Find zlib and JPEG!

In section 6.1.1 Installing zlib and section 6.1.2 Installing JPEG, I reported that I installed zlib and JPEG under `/usr/local`. The TK expects zlib and JPEG to be installed with the HDF4 library in the bowels of the TK subdirectories. There are two make files that had to be modified such that they could correctly reference zlib and JPEG; 1) the make file in `/opt/net/TOOLKIT/src/EPH/gbadsim`, and 2) the make file in `/opt/net/TOOLKIT/src/EPH/orbsim`. See the SARB Conversion Plan for the exact details.

#### 6.1.7.4 Installing the Toolkit Ancillary/Auxiliary Data Access Tools

The Toolkit Ancillary/Auxiliary (AA) Data Access Tools are *optionally* installed after the Toolkit has been installed. Even though an install script is provided in the Mac-Darwin distribution, the Toolkit installation script disables the AA install. After I modified the script to enable the AA tools installation I discovered why it had been disabled in the first place. The Mac-Darwin distribution includes an AA tools installation that has not been modified to be compatible with the Macintosh environment. In my first TK installation I suffered through the conversion of the script and the AA files so that they were



compatible with the Mac-Darwin environment. The conversion was difficult and lengthy but I documented every step in the SARB Conversion Plan. This turned out to be a fortunate decision on my part since the original installation of the Toolkit was destroyed in an operating system crash that required the Mac OS X operating system to be restored following a complete reformatting of the hard drive. It is my understanding that the AA tools are not used by the CERES subsystems, so in subsequent installations of the Mac-Darwin distribution of the TK I have skipped that laborious step<sup>92</sup>.

### 6.1.7.5 Where is malloc.h?

During the AA Data Access Tools conversion process described in section 6.1.7.4 Installing the Toolkit Ancillary/Auxiliary Data Access Tools, I made corrections to header file include statements that assumed the header file, `malloc.h`, was located in `/usr/include` with the rest of the standard C header files. In the Mac G5 Darwin configuration, `malloc.h` is located in `/usr/include/malloc` subdirectory. Later, I found this assumption occurs in other software products.

### 6.1.7.6 The Case of the Trailing Underscore

The first time I successfully completed installing the Toolkit, I immediately proceeded to the CERES library installation. When I tried to build the main CERES library, I encountered a problem with missing externals. Apparently, HDF4 implements a Fortran interface that assumes the Fortran compiler generates external names with a trailing underscore. By default, the IBM XLF compiler does not generate trailing underscores, so the CERES library references to the HDF4 modules were generated without trailing underscores. I added the `-qextname` compiler switch to the CERESlib make file, and I still had problems because the HDF4 interface had a mix of modules, some with trailing underscores and some without<sup>93</sup>. Consequently, I revisited the HDF4 build and added the `-qextname` compiler switch to the Fortran compiler settings. After thinking about this a while, I decided to rebuild the whole TK with consistent inclusion of the `-qextname` switch. Needless to say, I made the appropriate updates to the SARB Conversion Plan, and this too turned out to be fortuitous since I came back again and again to re-install the TK.

### 6.1.7.7 Toolkit User Accounts

Each user of the TK must be provided with a mechanism to define the many environment variables needed to compile, link, and run code with the TK. The Toolkit README file includes setup instructions for C-shell, Korn shell, and Bourne shell users. Other than make files, the CERESlib and SARB subsystem scripts were all C-shell scripts. On the Mac G5, the Panther 10.3.3 release defaults to the bash (Bourne again shell) shell for terminal windows. When I first started the conversion process, I elected to modify the Mac G5 shell script to the tcsh shell (an extension of the C-shell script, csh) in an effort to be directly compatible with the CERES C-shell scripts. Thus, I have configured the C-shell version of the Toolkit startup script, `pgs-dev-env.csh`. So, for the tcsh user on the Mac G5 the following line should be placed in the `.cshrc` file in the home directory:

---

<sup>92</sup> Lucky me! I have installed the Toolkit several times on each of the Mac G5 platforms that are supporting the SARB Conversion effort.

<sup>93</sup> This was due to the fact that some modules are explicitly named at the source level with the trailing underscore to facilitate being called by C procedures.

Source /opt/net/TOOLKIT/bin/macintosh/pgs-dev-env.csh

In section 3.1 What is Darwin, I illustrated how to change the default shell for Darwin.

## 6.2 The CERES Library Installation on the Mac G5

The CERES library implementation on the SGI platform maintains several versions of the same library due to the existence of more than one Fortran compiler and the fact that different subsystems use mutually exclusive compiler settings<sup>94</sup>. Since we are only using the IBM XLF compiler, we are not burdened with maintaining different compiler-dependent versions of CERESlib. Each impacted subsystem will be forced to resolve any such compiler issues during their respective conversions<sup>95</sup>. A positive result of this condition is that the CERESlib directory space is greatly simplified and much more compact on the G5. Due to a recent SARB release that required CERESlib updates, a more recent version of CERESlib has been installed on the Mac G5. The currently running version of CERESlib on the Mac G5 test platform is R3-560.

The CERESlib delivery package includes a fairly extensive test suite, so there are four phases to installing and converting the CERES library on the target platform; 1) install, convert and build the CERES library source code, run scripts and utilities, 2) install, convert, and build the test suite subdirectories with their test programs, run scripts, and expected value comparison files, 3) test the CERES libraries with the converted test suite software, and 4) test the CERES libraries indirectly while testing the SARB subsystem.

### 6.2.1 Installing the CERES Libraries

The CERESlib source code is structured such that there are only two source modules that need to be directly addressed for conversion to a new platform. The first module is named f90\_kind.f90, and the version I chose to include is from the 64-bit version of the SGI configuration. No modifications were required as the IBM XLF compiler KIND settings matched the SGI 64-bit KIND settings exactly. The second module is named ceres\_status.f90, and the version that I chose to include required one modification. The IBM XLF compiler generates I/O status code that returns a positive 1 for End-of-File status on a Direct Access file, and the SGI 64-bit version returns a negative 1. In case you overlook these two source modules, the CERES library test suite includes test programs that rigorously test the Kind and the various file status codes.

Once the source code is ready for compilation, the focus is on the scripts that control environment variable definitions and source code compilation. By necessity, CERESlib is the front line interface with the Toolkit, and so we see the emergence of a C-shell script that merges the CERES subsystem compiler and linker configuration settings with the Toolkit directory locations for libraries and related settings. In section 4.3.3 Build the Library, I listed the converted SARB version of this script (`ceres-env.csh`, see Listing 3 - CERES Environment Variable Definition Script). The `ceres-env.csh` script gets extensive modifications to convert it to the Mac-Darwin environment. The good news is that almost all of the remaining CERESlib C-shell scripts will run as is. In

<sup>94</sup> The best example of this is 32-bit versus 64-bit switch settings for the Fortran compiler.

<sup>95</sup> At first blush this is a scary statement but the IBM XLF compiler is a 64-bit compiler that can accommodate 32-bit applications. If we were going in the other direction (64-bit to 32-bit) I would be very concerned.

preparation to build the CERES library source files, the notable modifications to `ceres-env.csh` and the library make files were:

- Added `ADD_LFLAGS` and `ADD_LIBS` to `ceres-env.csh` to facilitate make file access to the zlib and JPEG libraries
- Added `-qextname` to the `F90COMP` and `FCOMP` compiler switch definitions in `ceres-env.csh` to force the IBM XLF compiler to generate trailing underscores on external names
- Added `-qsuffix=f=f90` to `F90COMP` in `ceres-env.csh` to force the IBM XLF compiler to use the file extension “.f90” for FORTRAN-90 source files
- Modified `CFLAGS` in `ceres-env.csh` to incorporate “-DMACINTOSH” setting for C functions that use conditional compilation based on the target platform type
- Replaced the library build command, “`ar rf $@ $?`” with “`ar rs $@ $?`” for compatibility with Darwin; the effected make files were:
  - `/CERES/lib/src/cereslib/Makefile`
  - `/CERES/lib/src/data_products/Makefile`
  - `/CERES/lib/src/write_data_files/common_lib/Makefile`
- Modified `F90LIB` in `ceres-env.csh` to `libxlf90.dylib` when `libxlf90.a` failed to link weak externals during the CERESlib build

In the attempt to build the CERES libraries and utility files, a few problems were revealed.

## 6.2.1.1 C Files Can't Find malloc.h

The CERES library includes some utility programs some of which are written in C. There were two occurrences of C header files that assumed `malloc.h` could be found in `/usr/include`. The Mac G5 Darwin configuration defines `malloc.h` in `/usr/include/malloc`. To resolve this problem the code in Listing 15 was added to two header files:

```
#ifdef MACINTOSH
#include <malloc/malloc.h>
#else
#include <malloc.h>
#endif
```

**Listing 15 - Conditional Compilation for malloc**

The modified header files are `/CERES/lib/src/bin_programs/Meta_read/Node.h` and `/CERES/lib/src/bin_programs/Meta_read_batch/Node.h`

### 6.2.1.2 Syntax Problem with ALLOCATABLE Array Declarations

A syntax error occurred during the compilation of `/CERES/lib/src/cereslib/meta_write.f90`. The error occurred for the declarations of the arrays named `CERGRingLon8`, `CERGRingLat8`, `lat_conv`, `lon_conv`, and `GRingSeq_Final`.

The syntax that caused the error is exemplified by Listing 16:

```
REAL(real8), DIMENSION(:), ALLOCATABLE :: CERGRingLon8, CERGRingLat8
REAL(real8), DIMENSION(:), ALLOCATABLE :: lat_conv, lon_conv
INTEGER,      DIMENSION(:), ALLOCATABLE :: GRingSeq_Final
```

Listing 16 - IBM XLF Syntax Errors

When I consulted the reference manual for the IBM XLF compiler, I was unable to determine what was wrong with the above syntax. The syntax errors were resolved by modifying the declarations to the equivalent syntax in Listing 17:

```
REAL(real8), ALLOCATABLE :: CERGRingLon8(:), CERGRingLat8(:)
REAL(real8), ALLOCATABLE :: lat_conv(:), lon_conv(:)
INTEGER,      ALLOCATABLE :: GRingSeq_Final(:)
```

Listing 17 - Revised Syntax for ALLOCATABLE Arrays

## 6.2.2 Installing the CERES Library Test Suite

The CERES library test suite software is distributed across 34 subdirectories in the path `/CERES/lib/test_suites`. Each subdirectory includes a make file for building the C or Fortran test code that also resides in the subdirectory. A few of the test suite subdirectories contain lower level subdirectories that contain test variations for their respective test functions. The `test_suites` parent directory contains a C-shell script (`makeall`) that, when invoked, iterates through all the `test_suite` subdirectories invoking the respective make files. During the attempt to build the `test_suite` subdirectories by executing the `makeall` script, there were a few immediate problems.

### 6.2.2.1 SGI-specific C Compiler Test Removed

The test in `/CERES/lib/test_suites/C_type_sizes` was removed from the Mac G5 test suite for the CERES library because it exercised tests that differentiated between 32-bit and 64-bit compiler configurations using the SGI C compiler. This test could be reengineered from scratch on the G5 platform but I found it expedient to take it out of play for the SARB conversion effort.

### 6.2.2.2 SGI-specific Compiler Defaults Test Removed

The test in `/CERES/lib/test_suites/Compiler_defaults` was removed from the Mac G5 test suite for the CERES library because it tests SGI Fortran and C, compiler-specific default settings with emphasis on 32-bit versus 64-bit compiler configurations. This test could be reengineered from scratch on the G5 platform but I found it expedient to take it out of play for the SARB conversion effort.

### 6.2.2.3 Unsatisfied External in Pcf\_c Test

During the attempt to build the CERES library test suite programs, the make file for /CERES/lib/test\_suites/Pcf\_c failed due to an unsatisfied external named “\_\_fill”. All attempts to identify the library containing \_\_fill failed, and I found it expedient to remove the Pcf\_c test and defer it until more effort could be expended to resolve the problem. The equivalent FORTRAN test suite, Pcf, did not present this problem.

### 6.2.2.4 Make File Bug Corrected

The make file in /CERES/lib/test\_suites/Defaults\_c had a typographical error that caused a make error during the test\_suite build. The first 3 lines of the make file (Makefile) are illustrated in Listing 18:

```

PROG = check_defaults.exe
SRCS = check_defaults.f90
OBJS = check_defaults.o

```

Listing 18 - Make File Typographical Error

The “SRCS” line was modified to:

```
SRCS = check_defaults.c
```

## 6.2.3 Testing the CERES Libraries with the Test Suite

The test suite parent directory contains a C-shell script (runtest) that, when invoked, will iterate over all the test suite subdirectories invoking their respective C-shell run scripts that are usually named “runtest”. The first attempt to run the test suite software produced two kinds of errors; 1) comparison errors that were due to acceptable differences, and 2) comparison errors that were due to legitimate errors.

### 6.2.3.1 We Agree to Differ

The most prominent acceptable difference originates from the comparison of the SGI Toolkit log files with their counterparts created by executing the test suite software on the Mac G5. Many of the test suite test programs call library functions that are dependent on the Toolkit. To use the Toolkit requires a Process Control File (PCF), and the PCF identifies the location and names for the standard Toolkit logs, Report, Status, and User. During the test, the three Toolkit log files are written into the respective “out\_comp” subdirectory, and when the test program completes, the log files in the out\_comp subdirectory are compared with their counterparts in the “out\_exp” (expected outputs) subdirectory. The expected output log files were created on the SGI platform using Toolkit version TK5.2.7, and the computed output log files are created on the Mac G5 platform using Toolkit version TK5.2.10. The Toolkit log maintenance software detects that the test programs are using a PCF from version TK5.2.7, and that the existing Toolkit is version TK5.2.10. This is duly noted in all three logs along with an 8-line warning message. The CERES library test suite comparison scripts report these comparison mismatches for all 3 log files for each occurrence. In every case, the test programs agree

on the actual subject matter under test but the comparison errors for the log files and Toolkit version discrepancy make the test appear to have catastrophically failed. This acceptable difference occurs in test subdirectories `Check_time`, `Constants`, `Defaults_c`, `Io/Open_da`, `Io/Read_nonexist`, `Io/Read_output`, `Io/Report_success`, `Io/Write_input`, `Meta_util`, `Msg/Test_report`, `Msg/Test_status`, `Pcf`, `Polar_flag`, `Reference_grid`, and `Solar_declination`. When I was sure that the impacted tests were working correctly and getting matching test results, I replaced the expected output log files with their counterparts from the G5 platform.

The `/CERES/lib/test_suites/Meta_util` test creates several `.met` files that are compared with files that were created on the SGI platform. The following comparison mismatch entry in Listing 19 is an example of an acceptable difference in the `.met` files:

```
521c521
<      VALUE   = "NASA Langley Research Center, HOST - thunder OS - IRIX64"
---
>      VALUE   = "NASA Langley Research Center, HOST - cts1-105 OS - darwin"
```

Listing 19 - Acceptable Comparison Mismatch

The `Meta_util` test also creates an HDF file that differs from the expected value HDF file. The HDF differences are in two of the binary components of the file, and they are the result of the Mac G5 using HDF5 as opposed to the SGI version that created the expected value file using HDF4.

The comparison differences in Listing 20 are taken from the `/CERES/lib/test_suites/Msg/Test_report` test:

```
56c65
< test_report: Less tokens than values ( 117.1) 22.
---
> test_report: Less tokens than values ( 117.1) 22.00000000
60c69
< test_report: No real tokens: 117.099998 22.
---
> test_report: No real tokens: 117.0999985 22.00000000
68c77
< test_report: Four real parameters without tokens: 117.09998 22. 3.5 4.250000045E-2
---
> test_report: Four real parameters without tokens: 117.099985 22.00000000 3.500000000
0.4250000045E-01
```

Listing 20 - Acceptable Comparison Differences

Note that within the single precision range, all the numbers are equivalent. The `/CERES/lib/test_suites/Msg/Test_status` test generates a similar set of comparison differences as shown in Listing 21:

```

22c31
< The bounding rectangle = ( 23.1, 4.1, 33.2, 13.3) 34.4000015
---
> The bounding rectangle = ( 23.1, 4.1, 33.2, 13.3) 34.40000153
28c37
< Message... The first number is 35, the 2nd number is 45.5999985 The 3rd number is 80.
---
> Message... The first number is 35, the 2nd number is 45.59999847 The 3rd number is
80.00000000

```

Listing 21 - More Acceptable Comparison Differences

Note that within the single precision range, all the numbers are equivalent.

### 6.2.3.2 Legitimate Test Suite Errors

Well, maybe the word legitimate is too strong. The test suite error described here is due to an error in test program `test_julian_date_routines.f90` in `/CERES/lib/test_suites/Ceres_time/Julian_date_routines`. The comparison output for this test looked catastrophically bad because the test output file was truncated due to the absence of a `CLOSE(11)` statement in the test program. After the `CLOSE(11)` statement was added to the test program, the comparison errors were resolved. But, this error teaches us that the SGI programs must get all their output files flushed at program end even though the output files may not have been closed. Apparently, this is not the case on the Mac G5 platform. This same error also occurred in program `test_julian_date_routines.f90` in subdirectory `/CERES/lib/test_suites/Ceres_time_c/Julian_date_routines`.

The `/CERES/lib/test_suites/Meta_read_batch` subdirectory does not contain a test program but it does implement a C-shell run-script that executes a utility program that is located at `/CERES/lib/bin`. The name of the program is `meta_read_batch.exe`, and it failed during the test suite execution phase. The `meta_read_batch` utility program is written in C and it gets compiled and linked during the build phase of the CERES libraries. The utility program was causing bus errors, and I was a little surprised to see that kind of problem during the test suite execution. However, when I started tracing through the source code I was no longer surprised. The `meta_read_batch` utility program is so poorly designed and written that I do not want to waste any more time on it. I was able to correct the coding errors so that the program would support the test suite execution without error, and I left it in that state. Quite frankly, I am surprised that it ever executed on the SGI platform, and I am heartened that it failed so miserably on the Mac G5. The `meta_read_batch` utility program badly needs to be redesigned and re-coded.

### 6.2.4 Testing CERESlib While Testing the SARB Subsystem

During the SARB subsystem conversion, there were some error conditions that were ultimately traced back to CERESlib.

### 6.2.4.1 The CERES Validation Regions Problem

During the testing of the SARB Main processor, CER5.1P1, I noticed that the QA-report file from the Mac G5 had zeroes in the Validation Regions part of the report in contrast to the QA-report from the SGI benchmark. I let this problem simmer on the back burner while I chased down more critical problems, and while I was chasing down other problems I became aware of a failure pattern that was developing on the Mac G5 platform. In two different cases I found that arrays that were initialized with DATA statements in the declaration part<sup>96</sup> of a MODULE returned zero values when they were referenced by code in a MODULE that was different from the declaring module. In each case, the referencing code was the only code to reference (USE) the arrays in question. In each case I used diagnostic code to prove that the referencing code was acquiring zero values from the arrays. No matter what I changed in the declaring MODULE, I could not alter the behavior of the anomaly. In each case, I resolved the problem by removing the declarative code from the owner module, and by placing the declaration in the MODULE that owned the referencing code. In both cases I retained the initialization code using the DATA statements, and in one of the cases I was forced to use a BLOCK DATA subprogram and a COMMON block.

When I finally got around to tracking down the Validation Regions data (all zeroes), I determined that the region numbers came from an array declared in the module `ceres_valregions.f90` in `/CERES/lib/src/cereslib`. The `ceres_valregions.f90` module declares and initializes several arrays including the CERES Validation Regions for SARB. All the arrays in `ceres_valregions.f90` are initialized by DATA statements, and the one that I was tracking was missing a PARAMETER modifier. I modified all the array initializations to use array constructor syntax rather than DATA statements, and I added the missing PARAMETER modifier. This resolved the immediate problem but I was left with two concerns here; 1) this may be an instance of the problem<sup>97</sup> I discussed above, and 2) the declaration part of `ceres_valregions.f90` is generated by a separate program, and thus it will be restored to the old form the next time `ceres_valregions.f90` gets updated.

### 6.2.4.2 The Case of the Missing Array Index

During the SARB testing I observed that both the SGI benchmark and the Mac G5 run-logs had non-fatal diagnostic messages that complained about not being able to find certain metafile objects. So, I decided to take a look and see if I could find the problem. The hunt for the problem turned up the code in `/CERES/lib/src/cereslib/meta_read.f90` as is shown in Listing 22:

```
DO I = 1, SIZE(Ivalue)
    Ivalue = INT4_DFLT
END DO
```

**Listing 22 - Missing Array Index**

<sup>96</sup> That is a declaration that occurs prior to the “CONTAINS” verb.

<sup>97</sup> I’m not sure if this problem is a bug in the IBM XLF compiler or not. I have tried unsuccessfully to create a “test tube” version of the problem using a small FORTRAN program. Thus, there is an indication that the problem is somehow context dependent.



I replaced `Ivalue` with `Ivalue(i)`, and I got no change in the problem that I was tracking. After I thought about this for a while I started to wonder if the original code was simply setting all the `Ivalue` array elements to `INT4_DFLT SIZE(Ivalue)` times, or was it setting the first element of `Ivalue` to `INT4_DFLT SIZE(Ivalue)` times?

### **6.3 The SARB Installation on Mac G5**

When porting software from one platform to another, the problem set can be broken into three general categories; 1) the build process better known as compile and link, 2) execution, and 3) verification of the results. For the SARB conversion effort, most of the difficulty was concentrated in the execution category as will be revealed in the following sections.

#### **6.3.1 SARB Library and PGE Compilations**

In section 3.5 The IBM XL FORTRAN Compiler (IBM XLF) I discussed my approach to the various compiler configurations that are necessary to “ring out” a complex program like any one of the SARB PGE’s. With the exception of setting the `-C` flag for the purpose of detecting array boundary violations, all the configurations generate the same results with regard to syntax errors and warning messages. The `-C` switch promotes the compile-time status of out of bounds array indices from warning to severe. In presenting the compile-time results from the SARB conversion, I am not going to attempt to distinguish the particular compiler switch configuration that was in effect when the error or warning was generated.

The SARB library constitutes the majority of the SARB subsystem source code. Thus, it makes sense that most of the compiler errors and warnings for the SARB subsystem occur during the library build process. Table 5 itemizes the warning messages for SARBlib, and Table 6 itemizes the SARBlib fatal compilation errors.

## CERES Conversion Guide

<i>File Name</i>	<i>Location Within File</i>	<i>Description</i>	<i>Recommendation</i>
<b><i>Makefile.CRS</i></b>	Dependencies	uvcor_all.o is dependent on uvcor_all.o (Circular dependency)	Remove uvcor_all.o from the list of dependencies for uvcor_all.o
<b><i>FL_Pass_Interface.f90</i></b>	Line 239	Redundant type definition for isky	Remove redundant type definition
<b><i>Tune_Code.f90</i></b>	Line 583	Literal string longer than variable length; literal truncated	Shorten literal string.
<b><i>IGBP_AdjSnowIce.f90</i></b>	Lines 245, 246, & 251	Redundant type definitions for asnow, asnow2, hcase	Remove redundant type definitions
<b><i>aotfit.f90</i></b>	Lines 88 and 145	Redundant type definitions for beta	Remove redundant type definitions
<b><i>Spectral_Sfc.f90</i></b>	Derived type declaration	7 misaligned variables; emiss, igbp1, ico, bba0, specalb, bba, specalb0	No recommendation.
<b><i>Makefile.CRS</i></b>	Dependencies	seiji_k2b.o is dependent on seiji_k2b.o (Circular dependency)	Remove seiji_k2b.o from the list of dependencies for seiji_k2b.o
<b><i>seiji_solver_0403.f90</i></b>	Line 119	Redundant type definitions for wc1, wc2, wc3, wc4, tt, and wc	Remove redundant type definitions
<b><i>seiji_solver_0403.f90</i></b>	Line 235	Redundant type definitions for wc1_c, wc2_c, wc3_c, wc4_c, tt_c, and wc_c	Remove redundant type definitions
<b><i>WindowFilter.f</i></b>	Line 257	Redundant type definition for trmm_fuliou_filter	Remove redundant type definitions

**Table 5 - SARBlib Compiler Warning Messages**

<i>File Name</i>	<i>Location Within File</i>	<i>Description</i>	<i>Action Taken</i>
<b><i>match_profiles.f90</i></b>	Declaration part of MATCH_PROFILES	In derived type declaration, MVTYPE, two character*7 arrays are declared using parameterized values for dimensions.	Changed parametric dimensions to literals
<b><i>misc_0403.f</i></b>	SR water_hu	The arrays tw, ww, and www are 1 <sup>st</sup> declared with dimensions, and then later are used in a COMMON statement with dimensions.	Removed dimensions from initial type statement
<b><i>seiji_twostreamsolv_sw_v20.f</i></b>	SR gwtsa_sw_v20	There are redundant dimension statements for w0_clear and g0_clear	Removed redundancy

**Table 6 - SARBlib Fatal Compiler Errors**

The IBM XLF compiler does not generate any warning messages or fatal compiler errors for the SARB Monthly Preprocessors (PGE CER5.0P1). The initial compilation of the SARB Main Processor (PGE CER5.1P1) generated a warning message that was

promoted to a fatal error when the -C compiler switch was specified. In the spirit of the tabular presentation of compiler errors we have Table 7 illustrating the singleton compiler snag for the SARB Main.

<i>File Name</i>	<i>Location Within File</i>	<i>Description</i>	<i>Action Taken</i>
<i>CRSDG_Output_Prep.f90</i>	SR Prep_Output_ CRSDG	Array CF is referenced by literal array indices that are outside the declared bounds	Modified the literal indices to be within bounds (code not currently used)

**Table 7 - SARB Main Processor Compiler Warning/Error**

Once the make file for CER5.3P1 was updated to include the HDF 5 libraries, a clean compile and link were achieved. CER5.4P1 consists of three executable programs in three separate directories. There were no fatal compiler errors for CER5.4P1 and Table 8 lists the compiler warning messages for all three executables.

<i>Directory/File Name</i>	<i>Location Within File</i>	<i>Description</i>	<i>Action Taken</i>
<i>hdf2crsb/hdf_read.f90</i>	Line 126	Identifier sd_id was previously defined with same type.	None
<i>qc_check/SARB_MonQC_WrapUp.f90</i>	Line 697	Truncated literal string	None
<i>qc_check/SARB_MonQC_HourProc.f90</i>	Line 183	Truncated literal string	None

**Table 8 - SARB QC Summary Processor Compiler Warnings**

### 6.3.2 SARB Runtime Issues

The two SARB PGE's that perform the mainstay of the processing burden are CER5.0P1 and CER5.1P1, the SARB Monthly Preprocessors and the SARB Main Processor. These two PGE's combined consist of three executable programs each of which is dependent on SARBlib. It should come as no surprise that most, if not all, of the runtime errors for CER5.0P1 and CER5.1P1 originate in SARBlib code. I debated on how to document the runtime errors for SARB; 1) in the chronological order of discovery, or 2) grouped according to their type. I chose the latter, and for groupings I decided to use the avoidable practices that I listed in section 0.

#### 6.3.2.1 SARB Issues with Mixed Mode Arithmetic

I am only documenting two mixed mode arithmetic issues for SARB but I believe the problem exists throughout the entire SARB library. For example, I believe that you could randomly choose a SARB subroutine that implements double precision arithmetic, and then starting in that subroutine trace backwards analyzing each variable and literal value that contributes to the value of each double precision variable used in the calculation. I am asserting that as you trace backward you will almost certainly find that 1) the calculation has mixed single and double precision variables, or 2) one or more of the double precision operands was computed with mixed mode arithmetic or was initialized with a single precision literal value. See section 0 for an example of mixed mode arithmetic. Table 9 lists the mixed mode issues discovered during the SARB conversion process. The first entry in the table is the example that is documented in section 0.

## CERES Conversion Guide

<i>Source File</i>	<i>Location Within File</i>	<i>Description</i>	<i>Action Taken</i>
<i>seiji_twostreamsolv_sw_v20.f</i>	Subroutine add	Double precision variable epsilon was initialized to single precision value 1.0E-15. Single precision variable du0 was used in mixed mode arithmetic to compute double precision variable c2. c2 was then used as an argument to single precision intrinsic, ABS and tested for .LE. with epsilon causing an inexact result for c2.	Made du0 double precision; initialized epsilon to 1.0D-15; calculated du0 as du0 = 1.0D0/DBLE(u0); used DABS on c2.
<i>seiji_twostreamsolv_sw_v20.f</i>	Entire Module	Mixed-mode arithmetic	Replaced single precision intrinsic calls with double precision intrinsics in real*8 calculations; added usage of DBLE to promote single precision variables to double precision when appropriate; added usage of SNGL to demote to single precision when appropriate; Modified single precision literals to double precision when they were used in real*8 calculations

**Table 9 - SARB Runtime Issues with Mixed Mode Arithmetic**

The second entry in the table is the result of a frustrating effort to resolve precision problems by removing the mixed mode arithmetic. This effort was not successful because it does not treat the single precision values that originate in other modules and flow toward the double precision computations that are performed in the module where the anomaly presents itself. I implemented the single module mixed mode cleanup on the SGI and the Mac G5, and the post-test result was a shift in the footprints that are rejected for Bad Tuned Cloud Fractions (BTCF's). By shift I mean that some footprints that were previously rejected for BTCF's were no longer rejected, and some footprints that were not previously rejected are now rejected. The overall number of BTCF's remained about the same, and these footprints were a fraction of a percent of the total number of footprints.

### 6.3.2.2 SARB Issues with Divide by Zero

The SARB library code has a healthy population of divide operations. Occasionally, prior to a divide operation you can find code that attempts to protect against a divide by zero. Unfortunately, this is the exception and not the rule. When I first became aware of this condition, I decided that I would retrofit all the subsystem code to protect against divide by zero. However, I quickly realized that I did not have time to undertake such an effort. In some cases there are several divide operators on the right hand side of a single assignment statement. Breaking up such code to check the divisors would be extremely invasive, and in some cases it would probably be unnecessary. I think the lesson learned here is that a subroutine, function, or procedure is on its way to being well behaved when it incorporates integrity checks on input parameters and divisor operands from the very beginning of its existence. This practice saves time during code development because the routine is self-diagnostic<sup>98</sup> as opposed to becoming a silent NaN generator. Self-diagnostic code is also a time saver when the code is ported to another platform. Table 10 illustrates the divide by zero events that caused the SARB software to trap when the compiler was configured to generate code that would trap on an invalid floating point operation signal from the floating point unit.

<i>File Name</i>	<i>Location Within File</i>	<i>Description</i>	<i>Action Taken</i>
<i>Spectral_Sfc.f90</i>	SR CldSnow	Variable AreaCld_Tot used as a divisor with value 0.0	Set variable Aprox_MeanTau_FOV to MaxTau when AreaCld_Tot == 0.0
<i>Spectral_Sfc.f90</i>	SR adj_spect_shape	Variable ss%bba(0) used as a divisor with value 0.0	Added code to detect ss%bba(0) == 0.0 and reset ss%bba(0) to 0.000001
<i>seiji_twostreamsolv_sw_v20.f90</i>	SR add	Divide by zero when double precision variable c1 is zero	Added code to detect c1 == 0.0 and set c1 = 1.0D-15
<i>TuneDrive.f90</i>	SR TunedSfcAlb	Div by zero when array bbout(P1) == 0.0	Added code to detect bbout(P1) == 0.0 and modify bbout(P1) = 1.0E-15
<i>seiji_solver_0403.f</i>	Fcn p_mle_seiji2	Div by zero when variable D is zero	When D == 0.0, set D to 1.0E-06
<i>misc_0403.f</i>	SR qftisf	Div by zero when variable ubr(i) == 0.0	Added code to detect ubr(i) == 0.0 and change ubr(i) to 1.00001
<i>misc_0403.f</i>	SR qftisf	Div by zero when variable xx == 1.0	Added code to detect xx == 1.0 and reset xx to 1.00001

Table 10 - SARB Divide by Zero Events

### 6.3.2.3 SARB Issues with Intrinsic Functions

This is another case when I am only documenting conditions that caused the processor to halt with a signal trap on an invalid floating point operation. In section 6.3.2.1 SARB Issues with Mixed Mode Arithmetic, I documented mixed mode arithmetic errors, and I

<sup>98</sup> Self-diagnostic is my term for a subroutine or function that will alert the programmer when something is wrong rather than have the problem lurking in the shadows for years possibly never being discovered.

mentioned that mixed mode problems existed throughout the SARB library. There are a fair number of cases where a single precision intrinsic function is called with a double precision argument, and there are also a fair number of cases where single precision operands are mixed with double precision operands with no usage of either DBLE or SNGL<sup>99</sup>. See section 0 for an example of this behavior. Table 11 illustrates the SARB errors involving intrinsic functions.

<i>File Name</i>	<i>Location Within File</i>	<i>Description</i>	<i>Action Taken</i>
<i>FL_Pass_Interface.f90</i>	SR FL_Call	LOG(0.0) and EXP(INF)	Added code to avoid LOG(0.0) and EXP(INF)
<i>aotfit.f90</i>	SR lsrsqr	SQRT(-X)	Added code to avoid SQRT of negative number <sup>100</sup>

**Table 11 - SARB Invalid Intrinsic Function Arguments**

#### 6.3.2.4 SARB Issues with Array Boundary Violations

Array boundary violations happen quietly unless you have deliberately attempted to screen them out by compiling all the code with signal trapping set and array bounds checking enabled. This particular avoidable condition is just as damaging to the output results as is mixed mode arithmetic. In particular, if the code is storing data into an array and the array index is out of bounds, then who knows where the stored data finds a home, and worse, when you are porting code the wayward data can and does find a home on the source platform and Murphy tells us that it will be a different address than the one found on the target platform. Table 12 illustrates the SARB array bounds violations.

<i>File Name</i>	<i>Location Within File</i>	<i>Description</i>	<i>Action Taken</i>
<i>Spectral_Sfc.f90</i>	SR as_ocean	Array AerConst_Angexp is referenced outside its declared bounds by variable DomAerType	Established a workaround for this problem but I never received corrective code from the SARB team. They told me the problem was of little scientific significance.
<i>Spectral_Sfc.f90</i>	SR snow_over	Array bounds violation when function ivector(ii) returns zero.	Checked for ivector(ii) returning zero and executed a return to avoid bounds violation <sup>101</sup>
<i>seiji_twostreamsolv_sw_v20.f</i>	SR raprad_twostr_sw_cloud	Two arrays with array bounds violations	Changed array indices to j1 from j – 1
<i>IGBP_AdjSnowice.f90</i>	SR CreateSphere	Code includes an	Array store was not

<sup>99</sup> Resistance is futile at this point since using DBLE to promote a single precision value to double precision will not add any more precision to the computation at hand.

<sup>100</sup> This event qualifies for inclusion in the section on unnecessary code since no code references the result.

<sup>101</sup> This is one of those cases that straddle two of my groupings. This case also belongs in the unnecessary code group.

<i>TuneDrive.f90</i>	SR Tune_Drv	array store that is outside of the array boundaries Array store outside of bounds for array ICnt_Tune_Err	needed; deleted  Modified QC_Init.f90 to change array bounds for ICnt_Tune_Err from 7 to 8. Also changed init code to init 8 elements to zeroes
<i>With_Cloud.f90</i>	SR CldLyr_ID	Undefined array element (NewLev_Idx(4)) used as array index to array ModLev causes array boundary violation	Reported error to SARB team and established a workaround fix by initializing NewLev_Idx to max legal value (31) <sup>102</sup>

Table 12 - SARB Array Bounds Violations

### 6.3.2.5 SARB Issues with Un-initialized Local Variables

My general impression is that the SARB code is pretty good about initializing local variables. I have only documented one case for this kind of problem and it almost slipped by without detection. See section 0 for this case.

### 6.3.2.6 SARB Issues with Executing Unnecessary Code

In some of the previous sections I have commented that the particular problem that I was describing also belongs in the section on executing unnecessary code. The reason for that is that in every case, I detected a problem because I was either trapping for array violations or invalid floating point operations. Invariably, when I would alert the SARB development team about my discovery, they would tell me that the real problem was that the code under scrutiny should not be executing under the current conditions. Table 13 illustrates the SARB issues with executing unnecessary code.

<i>File Name</i>	<i>Location Within File</i>	<i>Description</i>	<i>Action Taken</i>
<i>Spectral_Sfc.f90</i>	SR CldSnow	Call to Fcn flsasnow_lut returns MAX_REAL when nighttime flag is set and this causes INF's in SR adj_spectral_shape	Added code to detect when return value from flsasnow_lut is MAX_REAL; avoids call to SR adj_spect_shape with sfcalb0 set to MAX_REAL
<i>Spectral_Sfc.f90</i>	SR as_clr_land	Call to Fcn flsa_lut returns MAX_REAL when nighttime flag is set and this causes INF's in SR adj_spectral_shape	Added code to detect when return value from flsa_lut is MAX_REAL; avoids call to SR adj_spect_shape with sfcalb0 set to MAX_REAL
<i>FLSA_LUT_Utils.f90</i>	Fcn flsa_lut	Code executing when it	Added code to Fcn flsa_lut to

<sup>102</sup> The SARB team lead told me that 31 was not a good initial value. I did not argue the point but was more concerned about the underlying logic error in the subroutine that is responsible for assigning values to array NewLev\_Idx.



## CERES Conversion Guide

<i>FLSA_LUT_Utils.f90</i>	Fcn flsasnow_lut	should not be causes MAX_REAL value to be used in computations creating INF's Code executing when it should not be causes MAX_REAL value to be used in computations creating INF's	detect conditions indicating nighttime flag is set; return from flsa_lut when this occurs Added code to Fcn flsasnow_lut to detect conditions indicating nighttime flag is set; return from flsasnow_lut when this occurs
<i>Sfcalb_History.f90</i>	SR fill_sfcalb	Fcn ivector returns zero for lat and lon values over water. SR fill_sfcalb is for land computations and uses the zero returned from ivector to index an array outside its bounds	Added code to detect zero return from Fcn ivector and returned to caller in those cases

**Table 13 - SARB Issues with Executing Unnecessary Code**

### 6.3.2.7 SARB Mystery Anomaly

In section 6.2.4.1 The CERES Validation Regions Problem, I described a failure pattern that emerged during the SARB Main Processor conversion phase. In that section I referenced two occurrences of the failure pattern, and they are documented in Table 14.

<i>File Name</i>	<i>Location Within File</i>	<i>Description</i>	<i>Action Taken</i>
<i>Skdbl_ht02a.f90</i>	Entire MODULE	Module contains extensive DATA statements in Declaration part of the MODULE. References to the array values always return zeroes.	Moved contents of skdbl_ht02a.f90 to a BLOCK DATA unit at the end of file seiji_k2b.f90. Created Named COMMON block in BLOCK DATA unit and added Named COMMON block to code that references skdbl_ht02a.f90
<i>ZJIN_Params.f90</i>	Declaration part of MODULE	Declaration and initialization of array "anodes" incorporates extensive DATA statements in declaration part of MODULE. References to the array, anodes, always return zeroes.	Removed anodes declaration and initialization from ZJIN_Params.f90 and relocated them to ZJIN_Mod.f90

**Table 14 - SARB Mystery Anomaly**

I have created a "test tube" Fortran program that emulates the mystery anomaly, and I have not succeeded in recreating the failure. I could not find any statements in the IBM XLF compiler documentation that would prohibit usage of DATA statements in the declaration part of a MODULE, so it would appear that using the DATA statement to initialize arrays in the declaration part of a MODULE is perfectly acceptable. In both cases the DATA statements were broken up into array segments, and the number of array



elements was extensive. I checked for exceeded line lengths and for excessive line continuations, but found nothing in that area. Also, in both cases, the arrays in question were referenced by only one module. This fact allowed for the workaround fixes to succeed in both cases causing me to speculate that the problem may be a scoping issue that is somehow dependent on the linkage context for the SARB Main Processor. So, at this writing, the failure mode remains a mystery.

### 6.3.3 SARB Verification Results on Mac G5<sup>103</sup>

As documented in the SARB Conversion Plan, the SARB PGE's are delivered with test suite software that performs comparison functions on the respective SARB output products. The test suite software is designed to be deployed on the SGI platform for the determination of mismatches in the output records of two files; 1) the newly computed output file, and 2) the expected value file from a previous run on the test case in question. Thus, there is no functional test of the SARB software; you just determine whether there are mismatches between two files. This level of testing is perfect for converting software from one platform to another since we are assuming that the benchmark software is working just fine<sup>104</sup>, and we are not trying to update the software in the process of the conversion.

SARB PGE's CER5.0P1 and CER5.1P1 are the most difficult of the four PGE's. CER5.0P1 and CER5.1P1 produce the SARB output products while the other two PGE's CER5.3P1 and CER5.4P1 are involved with overhead operations; 1) converting from one file format to another file format, and 2) performing Quality Assurance operations on the output products. The verification results are reported here in the order of execution of the SARB PGE's.

#### 6.3.3.1 Verification Results for the SARB Monthly Preprocessors

The two SARB Monthly Preprocessors constitute CER5.0P1; 1) the Surface Albedo Monthly Preprocessor and 2) the Daily MODIS Aerosol Interpolation Monthly Preprocessor. CER5.0P1 is difficult to test because it requires so many large input files to be staged for a month of data. The test case that I chose for the SARB Monthly Preprocessors is defined by Table 15, environment variables for sampling strategies, production strategies, configuration codes, and the month of October 2001:

<i>Environment Variable</i>	<i>Value</i>	<i>Description</i>
<b>SAT</b>	Terra	Terra satellite
<b>INST</b>	FM2	CERES Flight Model 2 instrument
<b>IMAG</b>	MODIS	The MODIS imager
<b>SS5</b>	Terra-FM2-MODIS	Subsystem 5 sampling strategy
<b>SS4_5</b>	Terra-FM2-MODIS	Sampling strategy for subsystem 4 to subsystem 5
<b>SS12</b>	CERES	Sampling strategy for subsystem 12
<b>PS5</b>	Edition2B	Production strategy for subsystem 5
<b>PS4_5</b>	Edition2B	Production strategy for subsystem 4 to subsystem 5
<b>PS12</b>	DAO-GEOS4	Production strategy for subsystem 12
<b>CC5</b>	026030	Configuration code for subsystem 5
<b>CC4_5</b>	026030	Configuration code for subsystem 4 to subsystem 5

<sup>103</sup> Or, How I Learned to Love the Bomb! – Dr. Strangelove.

<sup>104</sup> I could not pass this opportunity by. It is a bad assumption.

<b>CC12</b>	016023	Configuration code for subsystem 12
<b>DATE</b>	200110	The Month of October in the year 2001

**Table 15 - SARB Monthly Preprocessor Test Case ID**

The full month test case executed on the Mac G5 in less than 45 wall clock minutes while the same benchmark case took 5 ½ wall clock hours on the SGI<sup>105</sup>. This is not a fair comparison because 1) the SGI platform was referencing the input files in the /DAAC directory structure and this slows down the I/O process greatly<sup>106</sup>, and 2) many other people were competing for resources when the benchmark case was executed<sup>107</sup>. For the Mac G5, all the input files were resident on the hard drive, and CER5.0P1 had few competing threads. The final comparison results speak for themselves and I have included them in Listing 23:

```

CER_HMPSAL_Terra-FM2-MODIS_Edition2B_026030.200110_test_suites_results
HOLD-2 MISMATCH—I=583092 File 1: 59 File 2: 60
TOTAL NUMBER OF MISMATCHES =      1

CER_HMSAL_Terra-FM2-MODIS_Edition2B_026030.200110_test_suites_results
HOLD-2 MISMATCH—I=583092 File 1: 59 File 2: 60
TOTAL NUMBER OF MISMATCHES =      1

CompareResults_CER_HMAER_Terra-FM2-MODIS_Edition2B_026030.200110
Total Number Mismatches:      0

diff CER_MQCSA_$INSTANCE /CERES/sarb/data/out_exp/data/sarb/CER_MQCSA_$INSTANCE
3c3
<          PAGE: 1          DATE PROCESSED: 12/13/2004 14:49:17
---
>          PAGE: 1          DATE PROCESSED: 12/13/2004 19:54:36

```

**Listing 23 - CER5.0P1 Comparison Results for 200110**

The single mismatch for the HMPSAL and the HMSAL output products is most likely the result of a rounding error since it differs by only 1 unit from the expected values computed on the SGI. Since there are literally millions of values being compared, I elected not to spend the effort to try to track down this single mismatch. I came to this conclusion after consultation with the SARB team revealed that there is no method for using the comparison output data to detect which one of the 744, 105Mbyte-SSFB<sup>108</sup> input files is the owner of the single data point in question. The HMPSAL and HMSAL data come from the SARB Surface Albedo Monthly Preprocessor, and the HMAER data comes from the Daily MODIS Aerosol Interpolation Monthly Preprocessor. The MQCSA data is the Quality Control report that provides statistics for the number of records available for each hour of each day of the month, and the two reports agree on

<sup>105</sup> No timing studies have been carried out on the PPC-970 cluster nodes because the PPC-970 cluster is still not available for test at this writing.

<sup>106</sup> Simply performing a directory list (ls) command in the directory containing the SSFB files can take several minutes due to the millions of files that populate the directory.

<sup>107</sup> I ran the benchmark case 3 times; 11 hours the 1<sup>st</sup> time, 6 ½ hours the 2<sup>nd</sup> time, and 5 ½ hours the 3<sup>rd</sup> time for an average of 7 hours and 40 minutes.

<sup>108</sup> Hourly Binary Single Satellite Footprint at 105 megabytes per file for this case.

everything but the processing dates which should not agree<sup>109</sup>. The four output products in Listing 23 each have a companion meta-file (.met). I performed comparisons on the .met files but I am not listing the results here since there were so many differences found. Each .met file stores the directory path and file name for the input files that contribute to the respective output product. The benchmark case from the SGI referenced the SSFB input files from the /DAAC directory on the SGI platform, and the Mac G5 stores the same SSFB input files under the /Inversion directory. Thus, all the directory paths were not in agreement between the SGI benchmark and the Mac G5 meta-files. All other object values in the meta-files compared with equality with the exceptions of processing dates, Toolkit versions, host platform (SGI versus Mac G5), and host operating systems.

### 6.3.3.2 Verification Results for the SARB Main Processor

CER5.1P1, the SARB Main Processor is an hourly processor, and in normal practice it executes once for each hour of a data month. CER5.1P1 was executed for the original 1-hour data case many times during the conversion process, and it consistently took 1 hour and 41 minutes of wall clock time when a safe level of optimization was configured during the build. The SGI wall clock time for the same test case was a consistent 3 ½ hours. At this writing, no timing studies have been performed on the DAAC cluster because it is not yet available for test. The original test case for CER5.1P1 is defined in Table 16.

<i>Environment Variable</i>	<i>Value</i>	<i>Description</i>
<b>SAT</b>	Terra	Terra satellite
<b>INST</b>	FM2	CERES Flight Model 2 instrument
<b>IMAG</b>	MODIS	The MODIS imager
<b>SS5</b>	Terra-FM2-MODIS	Subsystem 5 sampling strategy
<b>SS4_5</b>	Terra-FM2-MODIS	Sampling strategy for subsystem 4 to subsystem 5
<b>SS12</b>	CERES	Sampling strategy for subsystem 12
<b>PS5</b>	ValR2	Production strategy for subsystem 5
<b>PS4_5</b>	Edition2A	Production strategy for subsystem 4 to subsystem 5
<b>PS12</b>	DAO-GEOS4	Production strategy for subsystem 12
<b>CC5</b>	016020	Configuration code for subsystem 5
<b>CC4_5</b>	025029	Configuration code for subsystem 4 to subsystem 5
<b>CC12</b>	016023	Configuration code for subsystem 12
<b>DATE</b>	2001100101	Hour 1 of October 1, 2001

**Table 16 - CER5.1P1 Test Case Parameters**

Unfortunately the comparison results for CER5.1P1 were not as clean as those presented for CER5.0P1. The documented test case for 2001100101 includes 99,245 records, also known as footprints or fields of view. Out of the 99,245 footprints, there were comparison mismatches for a total of 123 footprints with 115 of the total footprints being rejected for Bad Tuned Cloud Fractions (BTCF's). On the Mac G5 side there were 91 footprints rejected for BTCF's, and on the SGI side there were 83 footprints rejected for BTCF's. Complicating the BTCF issue is the fact that the 91 Mac G5 footprints rejected for BTCF's are not a superset of the SGI set of 83 footprints rejected for BTCF's; rather,

<sup>109</sup> The arguments to the diff command include an environment variable, \$INSTANCE. In this case \$INSTANCE = "Terra-FM2-MODIS\_Edition2B\_026030.200110".

## CERES Conversion Guide

there is an intersection of 59 BTCTF-rejected footprints, so the G5 rejects 32 footprints that the SGI does not, and the SGI rejects 24 footprints that the G5 does not. When the two platforms do agree on a BTCTF-rejected footprint, the footprint still surfaces as a group of mismatches attributed to the same footprint in the comparison output<sup>110</sup>.

Consequently, each of the 115 BTCTF-rejected footprints requires more than a page of comparison output, therefore precluding a list of mismatches here. Listing 24 is a filtered version of the comparison mismatches for the CRSB<sup>111</sup> output file with all the BTCTF-rejected footprints removed.

CRS1= CRSB_Prod		CRS2= CRSB_CompareTo					
UpLW_TOA_CNASky_Delt		Rec #	23302	CRS1:	0.871735	CRS2:	2.080688
UpWN_TOA_CNASky_Delt		Rec #	23302	CRS1:	-0.012581	CRS2:	1.196388
UpLW_TOA_TotSky		Rec #	31749	CRS1:	1.219696	CRS2:	2.351410
Adj_Mn_CldTemp	Level 1	Rec #	31749	CRS1:	0.849731	CRS2:	1.072906
Adj_Mn_CldTemp	Level 2	Rec #	31749	CRS1:	0.985458	CRS2:	1.244705
UpLW_ClrSky	Level 1	Rec #	34129	CRS1:	245.413528	CRS2:	246.641815
UpWN_ClrSky	Level 1	Rec #	34129	CRS1:	86.245232	CRS2:	87.473541
UpLW_ClrSky	Level 2	Rec #	34129	CRS1:	246.376511	CRS2:	247.621918
UpWN_ClrSky	Level 2	Rec #	34129	CRS1:	87.802429	CRS2:	89.047798
UpLW_ClrSky	Level 3	Rec #	34129	CRS1:	250.199005	CRS2:	251.461380
UpWN_ClrSky	Level 3	Rec #	34129	CRS1:	90.749603	CRS2:	92.011963
UpLW_ClrSky	Level 4	Rec #	34129	CRS1:	286.584961	CRS2:	287.858673
UpWN_ClrSky	Level 4	Rec #	34129	CRS1:	93.104561	CRS2:	94.378288
UpLW_TOA_ClrSky		Rec #	34129	CRS1:	-1.604187	CRS2:	-0.375961
UpWN_TOA_ClrSky		Rec #	34129	CRS1:	-1.593483	CRS2:	-0.365257
UpLW_TOA_ClrSky		Rec #	34676	CRS1:	3.401230	CRS2:	0.831802
UpWN_TOA_ClrSky		Rec #	34676	CRS1:	3.400566	CRS2:	0.831230
UpLW_PrsSky	Level 1	Rec #	35156	CRS1:	245.425400	CRS2:	244.446228
UpSW_TotSky	Level 1	Rec #	54334	CRS1:	118.011826	CRS2:	116.989044
UpLW_TotSky	Level 1	Rec #	54334	CRS1:	172.026489	CRS2:	176.169128
UpWN_TotSky	Level 1	Rec #	54334	CRS1:	42.174004	CRS2:	44.258781
UpSW_TotSky	Level 2	Rec #	54334	CRS1:	114.994942	CRS2:	113.970009
UpLW_TotSky	Level 2	Rec #	54334	CRS1:	171.724091	CRS2:	175.922028
UpWN_TotSky	Level 2	Rec #	54334	CRS1:	42.431015	CRS2:	44.565685
UpSW_TotSky	Level 3	Rec #	54334	CRS1:	108.353584	CRS2:	107.323029
UpLW_TotSky	Level 3	Rec #	54334	CRS1:	171.898010	CRS2:	176.240707
UpWN_TotSky	Level 3	Rec #	54334	CRS1:	43.270378	CRS2:	45.512897
UpSW_TotSky	Level 4	Rec #	54334	CRS1:	94.231133	CRS2:	93.107452
UpLW_TotSky	Level 4	Rec #	54334	CRS1:	203.691528	CRS2:	208.724335
UpWN_TotSky	Level 4	Rec #	54334	CRS1:	48.876091	CRS2:	51.210129
UpSW_TOA_TotSky		Rec #	54334	CRS1:	6.651497	CRS2:	5.628716
UpLW_TOA_TotSky		Rec #	54334	CRS1:	-18.936371	CRS2:	-14.793701
UpWN_TOA_TotSky		Rec #	54334	CRS1:	-10.189011	CRS2:	-8.104229
UpSW_CNASky	Level 1	Rec #	54334	CRS1:	118.192314	CRS2:	117.138016
UpLW_CNASky	Level 1	Rec #	54334	CRS1:	172.089584	CRS2:	176.240326
UpWN_CNASky	Level 1	Rec #	54334	CRS1:	42.214703	CRS2:	44.304871
UpSW_TOA_CNASky_Delt		Rec #	54334	CRS1:	6.785408	CRS2:	5.731117
UpLW_TOA_CNASky_Delt		Rec #	54334	CRS1:	-18.973053	CRS2:	-14.822372
UpWN_TOA_CNASky_Delt		Rec #	54334	CRS1:	-10.214741	CRS2:	-8.124619
Constraintment Flag Mismatch		Rec #	58488	CRS1:	0	CRS2:	100
Pressure Level	Level 4	Rec #	62505	CRS1:	500.000000	CRS2:	498.000061
TOTAL RECORD MISMATCHES:		8					
RECORDS SKIPPED FOR BAD TUNED CLOUD FRACTIONS:		115					

**Listing 24 - CER5.1P1 Comparison Mismatches with Filtered BTCTFs**

The comparison software generally compares two values by taking the absolute value of their difference and checking the result for greater than 0.1. None of the comparison mismatches in Listing 24 is catastrophically bad, and in total they constitute only .0081%

<sup>110</sup> So, even when both platforms reject a footprint for BTCTF, they still don't agree on the footprint values.

<sup>111</sup> Cloud Radiative Swath Binary, the principle output file for CER5.1P1.

## CERES Conversion Guide

of the 99,245 footprints<sup>112</sup>. Combined with the footprints that are rejected for BTCF's, the total comparison mismatches effect only .124% of the 99,245 footprints.

CER5.1P1 also outputs a meta- file for the CRSB, the CRSVB, and the HQCR output files. I performed comparisons on the .met files but I am not listing the results here since there were so many differences found. Each .met file stores the directory path and file name for the input files that contribute to the respective output product. The benchmark case from the SGI referenced the input files relative to my home directory on the SGI platform, and the Mac G5 references the input files relative to the /CERES/sarb directory. Thus, all the directory paths were not in agreement between the SGI benchmark and the Mac G5 meta- files. All other object values<sup>113</sup> in the meta- files compared with functional equality with the exceptions of processing dates, Toolkit versions, host platform (SGI versus Mac G5), and host operating systems.

Finally, the comparison between the SGI QC report and the Mac G5 QC report also has many comparison mismatches. Listing 25 illustrates the 1<sup>st</sup> several entries in the comparison output.

3c3					
<	PAGE: 1		DATE PROCESSED: 11/03/2004 12:12:02		
---					
>	PAGE: 1		DATE PROCESSED: 10/18/2004 15:35:27		
43c43					
<	INVALID SW UPWARD	PRISTINE SKY - INITIAL	:	3	
---					
>	INVALID SW UPWARD	PRISTINE SKY - INITIAL	:	5	
45c45					
<	INVALID SW UPWARD	CLEAR SKY - INITIAL	:	1	
---					
>	INVALID SW UPWARD	CLEAR SKY - INITIAL	:	0	
117c117					
<	INVALID SW UPWARD	PRISTINE SKY - CONSTRAINED	:	0	
---					
>	INVALID SW UPWARD	PRISTINE SKY - CONSTRAINED	:	1	
185c185					
<	NUMBER OF BAD TUNED CLOUD FRACTIONS		:	91	
---					
>	NUMBER OF BAD TUNED CLOUD FRACTIONS		:	83	
193,194c193,194					
<	0 CLOUD LAYERS :	0			
<	1 CLOUD LAYERS :	93			
---					
>	0 CLOUD LAYERS :	1			
>	1 CLOUD LAYERS :	85			
197,198c197,198					
<	0 CLOUD LAYERS :	4896			
<	1 CLOUD LAYERS :	23092			
---					
>	0 CLOUD LAYERS :	4895			
>	1 CLOUD LAYERS :	23099			

**Listing 25 - CER5.1P1 Excerpt of Comparison Output for QC Report**

In Listing 25 the comparison differences for the “INVALID SW UPWARD...” entries are all due to both the SGI and the G5 finding out of range flux values at the last layer of the modeled atmosphere. In every case the out of range values are small negative numbers

<sup>112</sup> For these 8 footprints the percentage is computed by  $100 * 8 / 99245$ . So, I am being conservative because there are many data objects in a footprint and each object is compared. My computation takes the whole footprint out of play when just one object fails the comparison criteria with the 0.1 delta!

<sup>113</sup> The meta-files do contain floating point data and there are many differences between the SGI and the G5 but they are all equivalent values within the single precision range. Some of the differences are based on the comparison of numbers like 180.0 and 180.000000.

that are in the neighborhood of 1E-15. When you inspect the footprints involved in these mismatches, the rest of the flux values in the lower layers compare nicely within single precision range of 6 or 7 digits. So, in these cases the respective footprints are being rejected because of a single flux computation that is essentially zero. I believe that this condition is a product of the precision problem that stems from the use of mixed mode arithmetic in the subroutines that are used to compute the short wave (SW) flux values (see section 0). Listing 26 illustrates a typical entry in the latter portion of the QC report file comparison data.

ID#	Description		Avg	StdDev	Good#	Skew	Kurt	Min	Max
460,462c460,462									
< 150 CLEAR SFC	NET DIV		512.121	247.762	4895	-0.439	-0.047	-190.404	933.630
< 151 CLEAR ORIG	SKINT		292.496	13.922	9118	-0.527	1.287	236.375	328.706
< 152 CLEAR ADJUST	SKINT		0.100	1.234	9118	1.110	5.251	-8.787	7.789
---									
> 150 CLEAR SFC	NET DIV		512.120	247.762	4895	-0.439	-0.047	-190.404	933.630
> 151 CLEAR ORIG	SKINT		292.496	13.922	9118	-0.528	1.327	236.375	328.706
> 152 CLEAR ADJUST	SKINT		0.101	1.234	9117	1.111	5.255	-8.787	7.789
464c464									

**Listing 26 - CER5.1P1 Typical Entry in QC Report Comparison Output**

I spent several months testing the SARB Main Processor, CER5.1P1, and I rigorously screened the code for invalid floating point operations and out of range array references. Each time I found problematic code I corrected the source on the G5 and on the SGI, and I produced new results on both platforms. I believe that the remaining differences in the output data are due to differences in, or the lack of, precision between the two platforms. I spent almost 3 weeks analyzing the computations involved in a footprint that gets rejected for Bad Tuned Cloud Fraction, and all I could find were some differences in precision in the initial flux values. I could never find a “smoking gun” condition where the SGI platform “zigged” when the G5 was “zagging”. However, I did find that the code that produces the flux values is based on mixed mode arithmetic. I am now at the point where I believe that no more effort should be spent trying to narrow the differences until the SARB CER5.1P1 code is upgraded to be rid of the usage of mixed mode arithmetic.

### 6.3.3.3 Verification Results for the SARB Postprocessor

CER5.3P1 is called a postprocessor because it executes once following each hourly execution of the main processor (CER5.1P1). In production mode the postprocessor executes once for each hour of data and its sole purpose is to convert the hourly CRSB file to its HDF equivalent. In the SARB Conversion Plan I planned to test the postprocessor as a standalone program but I was unable to conveniently verify the HDF file because the SARB system performs the HDF verification during the execution of CER5.4P1, the QC Summary processor. Also, the HDF verification software is designed to only verify a maximum of five HDF files for a given data month<sup>114</sup>. Consequently, I executed two of the five verification test cases so there would be two HDF files for verification.

The method of verification is not dependent on generating the same HDF files on the source platform because any differences in the CRSB files would be retained in the HDF

<sup>114</sup> The dates for the 5 verification files are YYYYMM0106, YYYYMM0809, YYYYMM1415, YYYYMM2118, and YYYYMM3023, where YYYY are 4 digits for the year of the data case, and MM is the month of the data case.

files. The way the SARB team verifies and HDF is to convert it back to a CRSB file and then compare the resultant CRSB file with the original that was input into the CER5.3P1 postprocessor. There should be no differences. The SARB PGE, CER5.4P1 invokes two programs to perform the HDF verification. The first program creates a CRSB file from the HDF file and the second program compares the resultant CRSB files with the CRSB files that were originally used as input to CER5.3P1. During this process the comparison results are temporarily stored in a text file, and then at the conclusion of the CER5.4P1 run-script, the comparison results text is E-mailed to the SARB team lead. I modified the run-script to send the E-mail to my E-mail address, and Listing 27 is the E-mail that I received as a result of executing the CER5.4P1 run-script.

```
To: j.l.donaldson@larc.nasa.gov
Date: Thu, 6 Jan 2005 12:32:33 -0500 (EST)
From: Donaldsn@CTS1-105.local (James Donaldson)

Terra-FM2-MODIS_Edition2B_026030.2001100106
TOTAL RECORD MISMATCHES: 0
Terra-FM2-MODIS_Edition2B_026030.2001100809
TOTAL RECORD MISMATCHES: 0

Terra-FM2-MODIS_Edition2B_026030.200110
SARB POST CRSBCOMP PGE EXIT = 0
```

**Listing 27 - HDF Verification Results Via E-mail**

I would have executed all five of the verification dates but I had only acquired the first eight full days of SSFA files for the data month under test (October 2001). The appropriate hourly SSFA file is required as an input file when the CRSB file is created from the HDF file.

#### 6.3.3.4 Verification Results for the SARB QC Summary Processor

The SARB QC Summary processor verifies five of the possible 744 HDF files as described in the last section. The summary processor acquires QC summary data by reading and parsing each of the possible 744 hourly QC files. Once the summary QC data is acquired the processor outputs 3 tabular reports. The tabular reports are stored at \$CERESHOME/sarb/data/out\_comp/qa\_reports/sarb. Table 17 describes the formats.

<i>File Name</i>	<i>Format</i>	<i>Description</i>
<b>CER_HMRV_\$INSTANCE</b>	ASCII text	Monthly CERES Region Report
<b>CER_HMRV_\$INSTANCE.html</b>	Browser-ready text file	Monthly CERES Region Report
<b>CER_HMQCR_\$INSTANCE</b>	ASCII text	Monthly QC Report
<b>CER_HMQCR_\$INSTANCE</b>	Browser-ready text file	Monthly QC Report
<b>CER_HMAVAIL_\$INSTANCE</b>	ASCII text	Availability table

**Table 17 - CER5.4P1 Output Files and Formats**

I verified the various tables by inspection since I had only provided a total of three hourly test cases prior to running CER5.4P1. Three test cases cause the summary tables to be sparse but it is enough to verify the proper operation of the QC Summary processor. The latest version of CER5.4P1 also uses IDL to create statistical graphics files. I did not receive the IDL version until very late in the conversion process, so I bypassed the IDL part of the run-script for CER5.4P1. The IDL functionality will be re-visited and this document will be updated to reflect the results.

## CERES Conversion Guide

The CER5.4P1 run-script sends the HMRV, HMQCR, and the HMAVAIL tables to the SARB team lead in three E-mail messages. I modified the run-script to send the E-mail messages to me and I have included all of or portions of the messages here. Listing 28 is the CERES Region Report.

To: [j.l.donaldson@larc.nasa.gov](mailto:j.l.donaldson@larc.nasa.gov)  
Date: Thu, 6 Jan 2005 12:32:00 -0500 (EST)  
From: [Donaldsn@CTS1-105.local](mailto:Donaldsn@CTS1-105.local) (James Donaldson)

### Monthly CERES Region Report

Data Set Descriptors:  
YYYY-MM: 2001-10  
Sampling Strategy: Terra-FM2-MODIS  
Production Strategy: Edition2B  
Configuration Code: 026030

RegNum days/hrs... passed over

01	01/07	08/10
3789	08/10	
3793	08/10	
4149	08/10	
4153	08/10	
4189	01/07	08/10
6501	01/07	08/10
6505	01/07	
6861	01/07	08/10
6865	01/07	
9199	08/10	
11791	01/07	
12149	01/07	
14807	01/07	
14817	01/07	
16258	01/07	
16259	01/07	
16261	01/07	
16271	01/07	
16716	05/24	
17345	01/07	
18425	01/07	
18619	01/07	
18784	01/07	
18785	01/07	
18878	05/24	



## CERES Conversion Guide

19144	01/07
19145	01/07
19505	01/07
19860	01/07
19862	01/07
19864	01/07
19874	01/07
20218	01/07
20219	01/07
20220	01/07
20221	01/07
20222	01/07
20226	01/07
20229	01/07
20234	01/07
20580	01/07
20581	01/07
20582	01/07
20594	01/07
20758	01/07
20940	01/07
20941	01/07
20942	01/07
20951	01/07
22550	01/07
22551	01/07
22552	01/07
22911	01/07
22912	01/07
23271	01/07
24143	08/10
24144	08/10
24504	08/10
24505	08/10
24862	08/10
24863	08/10
24864	08/10

## CERES Conversion Guide

24865	08/10
25223	08/10
25224	08/10
25225	08/10
25717	05/24
25718	05/24
25719	05/24
26076	05/24
26077	05/24
26078	05/24
26079	05/24
26436	05/24
26437	05/24
26438	05/24
26439	05/24
26797	05/24
26798	05/24
26799	05/24
26802	05/24
26803	05/24
26804	05/24
26805	05/24
27162	05/24
27163	05/24
27164	05/24
27165	05/24
27522	05/24
27523	05/24
27524	05/24
27525	05/24
27883	05/24
27884	05/24
30486	01/07
30756	05/24
34725	05/24
34726	05/24

## CERES Conversion Guide

34727	05/24	
35085	05/24	
35086	05/24	
35087	05/24	
35088	05/24	
35445	05/24	
35446	05/24	
35447	05/24	
35448	05/24	
35805	05/24	
35806	05/24	
35807	05/24	
37090	08/10	
37450	08/10	
48949	05/24	
50439	08/10	
57099	05/24	
57101	05/24	
57459	05/24	
57461	05/24	
57769	05/24	
57773	05/24	
58129	05/24	
58133	05/24	
60461	05/24	
64441	05/24	08/10
Date Report Generated: 20050106 123200.294		

### Listing 28 - Monthly CERES Region Report

Each of the three test hours can be found associated with their respective region numbers and these instances can be cross-checked with the QA reports from each hourly test case.

Listing 29 is the Monthly CERES CRS Hour Availability Table, and I have highlighted in red the hourly cases that were marked as available.

# CERES Conversion Guide

To: [j.l.donaldson@larc.nasa.gov](mailto:j.l.donaldson@larc.nasa.gov)  
 Date: Thu, 6 Jan 2005 12:32:00 -0500 (EST)  
 From: [Donaldsn@CTS1-105.local](mailto:Donaldsn@CTS1-105.local) (James Donaldson)

Terra-FM2-MODIS\_Edition2B\_026030.200110  
 SARB POST MQC PGE\_EXIT = 0

Please retrieve the tar file and post the Plots and Reports

## Monthly CERES CRS Hour Availability Table

Data Set Descriptors:  
 YYYY-MM: 2001-10  
 Sampling Strategy: Terra-FM2-MODIS  
 Production Strategy: Edition2B  
 Configuration Code: 026030

Hour	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Day 01	u	u	u	u	u	u	A	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 02	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 03	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 04	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 05	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	A
Day 06	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 07	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 08	u	u	u	u	u	u	u	u	u	A	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 09	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 10	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 11	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 12	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 13	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 14	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 15	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 16	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 17	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 18	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 19	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 20	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 21	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 22	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 23	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 24	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 25	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 26	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 27	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 28	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 29	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 30	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u
Day 31	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u

Date Report Generated: 20050106 123200.295

**Listing 29 - Monthly CERES CRS Hour Availability Table**

The three highlighted cases correspond correctly with the cases that I made available for testing the QC Summary postprocessor.

## CERES Conversion Guide

The HMQCR report is the most lengthy of the three reports, and it is too long to list in this context. In verifying the non-zero entries in the Monthly QC Report, I found one anomaly. I looked at the SGI Monthly QC Report for the same data month, and I found that the SGI report has the same anomaly. Listing 30 illustrates two excerpts from the Monthly QC Report from my test run. I have highlighted in red the single entry for one of the three hourly cases that I used. Inspection of the QA report for Day 1, hour 6

❏❏❏ snip ❏❏❏❏							
Tuning Error Statistics							
Hour		00	01	02	03	04	05
Day 01	SIGTAB XCD MAXTUNE	0	0	0	0	0	0
Day 01	SIGTAB XCD MCLD	0	0	0	0	0	0
Day 01	SIGTAB XCD NSID	0	0	0	0	0	0
Day 01	OCCRS SBR TUNE_XXX>10	0	0	0	0	0	0
Day 01	CLD FRAC ADJ ERRS	0	0	0	0	0	0
Day 01	SINGULAR MATRIX ERRS	0	0	0	0	0	0
Day 01	BAD TUNED CLD FRACS	0	0	0	0	0	0
Day 02	SIGTAB XCD MAXTUNE	0	0	0	0	0	0
Day 02	SIGTAB XCD MCLD	0	0	0	0	0	0
Day 02	SIGTAB XCD NSID	0	0	0	0	0	0
Day 02	OCCRS SBR TUNE_XXX>10	0	0	0	0	0	0
Day 02	CLD FRAC ADJ ERRS	0	0	0	0	0	0
Day 02	SINGULAR MATRIX ERRS	0	0	0	0	0	0
Day 02	BAD TUNED CLD FRACS	0	0	0	0	0	0
❏❏❏ snip ❏❏❏❏							
Hour		06	07	08	09	10	11
Day 01	SIGTAB XCD MAXTUNE	141	0	0	0	0	0
Day 01	SIGTAB XCD MCLD	0	0	0	0	0	0
Day 01	SIGTAB XCD NSID	0	0	0	0	0	0
Day 01	OCCRS SBR TUNE_XXX>10	0	0	0	0	0	0
Day 01	CLD FRAC ADJ ERRS	0	0	0	0	0	0
Day 01	SINGULAR MATRIX ERRS	0	0	0	0	0	0
Day 01	BAD TUNED CLD FRACS	0	0	0	0	0	0
❏❏❏ snip ❏❏❏❏							

**Listing 30 - Tuning Error Statistics Anomaly**

reveals that there were 141 BAD TUNED CLD FRACS and zero occurrences of SIGTAB XCD MAXTUNE. This is contrary to the summary report shown in Listing 30. Both the SGI and the Mac G5 Monthly QC Reports consistently misplace the BAD TUNED CLD FRACS counters. I found no other anomalies.

## 7 Lessons Learned From the SARB Conversion

Prior to converting the SARB subsystem, I had to install the PGS Toolkit and CERESlib. I learned a lot about the IBM XLF compiler and the Mac G5 platform but I have to admit, the SARB conversion was an even greater learning process. It is a difficult task to try to separate the generic lessons learned from the familiarization process that occurs when working with the same software for several months. The question that arises is, given a new subsystem the same size as SARB with the same level of complexity, would I be able to perform the conversion much faster because of the lessons learned from the SARB conversion effort? I assert that the answer is in the affirmative if you allow for the time that it takes to become familiar with the new code. If the new code follows the same coding standard, follows the same generic directory mapping scheme, and is compliant with the CERES data management guidelines<sup>115</sup>, then I should not care if I'm converting SARB or Clouds.

### 7.1 Adopt a Coding Standard and be Consistent



**Mac says 10 - As software development budgets continue to come under pressure for all sorts of reasons, it becomes painfully clear that it is prohibitively expensive to revisit badly written code to make it readable and maintainable.**

I don't believe that a coding standard should be so constrictive that the programmers feel like they are overly constrained. I do believe that the development teams should agree on a flexible coding standard that incorporates guidelines on things like 1) variable naming, 2) indentation for block statements<sup>116</sup>, 3) a minimum level and style for internal documentation (commenting code), 4) consistent usage of headers for block data subprograms, functions, subroutines, programs and modules, 5) making an effort to adhere to a language standard like FORTRAN 95, and 6) generally following good programming practices.

In working with the SARB subsystems, it becomes quickly evident that there is a continuing effort to clean up the code. I am not trying to criticize a work in progress; rather I am suggesting that certain coding practices never make their way into production code from the very start. As software development budgets continue to come under pressure for all sorts of reasons, it becomes painfully clear that it is prohibitively expensive to revisit badly written code to make it readable and maintainable. Where does the pain come into the picture? After all the code works, does it not? The pain enters the picture when the badly written code fails during an attempt to migrate the code to a new platform. Then, you can spend hours instead of minutes determining what should be done. The pain also enters the picture when you decide that you want to provide your code to a service provider who maintains a high degree of integrity on the target system. For example, the service provider might not allow code that generates invalid floating point results like NaN's and INF's.

<sup>115</sup> I was unsuccessful in finding the CERES data management guidelines document.

<sup>116</sup> DO, DO-WHILE, IF-THEN, CASE, E.T.C.

### 7.1.1 Comply with the FORTRAN Standard

If the CERES subsystem developers had strictly complied with the FORTRAN 90 standard, I would not be writing this document. This is not a fair statement to make because a lot of the CERES science code had its beginning prior to the publication of the FORTRAN 90 standard, not to mention the FORTRAN 95 standard. So, how does one make legacy code compliant with a standard that is younger than the code? The realistic answer is that it probably will not happen except for new software development. What is realistic is to perform an assessment of the subsystem code, and then identify and prioritize the improvements that have the highest positive impact on the program. One example that comes to mind is the recommendation of the FORTRAN standards committee to move away from the usage of COMMON blocks by declaring the variables to be shared in MODULEs. I think this should be a high priority consideration for the CERES subsystems if it is done correctly.

While converting the SARB Main Processor (CER5.1P1) I was intimidated by the bulk of data that is defined using DATA statements. The data was then frequently accessed by functions and subroutines via named COMMON blocks. I am not exaggerating when I

```
data ((sktbl02(ih,          11,          2,ik),ik=1,07),ih=1,13)/&
2.134E-17,1.502E-15,1.265E-14,5.815E-14,6.125E-13,7.284E-11,2.136E-08,&
8.396E-17,2.856E-15,2.654E-14,1.573E-13,1.853E-12,2.123E-10,6.481E-08,&
4.866E-16,7.119E-15,6.885E-14,5.440E-13,6.497E-12,6.590E-10,2.016E-07,&
2.843E-15,2.735E-14,2.851E-13,2.399E-12,2.807E-11,2.188E-09,6.239E-07,&
1.828E-14,1.738E-13,1.833E-12,1.530E-11,1.669E-10,8.541E-09,1.976E-06,&
1.123E-13,1.369E-12,1.481E-11,1.221E-10,1.237E-09,3.906E-08,6.175E-06,&
9.177E-13,1.258E-11,1.410E-10,1.130E-09,1.077E-08,2.202E-07,2.026E-05,&
7.457E-12,1.182E-10,1.344E-09,1.067E-08,9.478E-08,1.382E-06,7.069E-05,&
6.670E-11,1.173E-09,1.345E-08,1.064E-07,8.810E-07,9.725E-06,2.875E-04,&
6.200E-10,1.110E-08,1.301E-07,1.020E-06,7.738E-06,6.963E-05,1.330E-03,&
5.846E-09,1.074E-07,1.281E-06,9.837E-06,6.362E-05,5.008E-04,6.562E-03,&
5.425E-08,9.753E-07,1.207E-05,8.230E-05,4.284E-04,3.074E-03,2.824E-02,&
5.434E-07,9.475E-06,1.095E-04,5.780E-04,2.604E-03,1.705E-02,1.212E-01 /
! SKIP          2          1          3          0
! SKIP          2          2          3          0
! SKIP          2          3          3          0
data ((sktbl02(ih,          4,          3,ik),ik=1,07),ih=1,13)/&
1.519E-16,2.039E-14,3.307E-13,1.637E-12,8.299E-12,7.870E-11,7.981E-09,&
7.580E-16,6.009E-14,6.703E-13,3.501E-12,1.684E-11,2.150E-10,2.268E-08,&
4.118E-15,1.817E-13,1.586E-12,7.544E-12,4.121E-11,6.188E-10,6.846E-08,&
2.109E-14,5.744E-13,4.051E-12,1.862E-11,1.200E-10,1.869E-09,2.074E-07,&
1.409E-13,1.958E-12,1.291E-11,7.434E-11,4.167E-10,6.361E-09,6.470E-07,&
1.163E-12,1.334E-11,5.987E-11,3.758E-10,1.803E-09,2.552E-08,2.063E-06,&
1.073E-11,1.194E-10,4.326E-10,2.661E-09,1.147E-08,1.440E-07,7.216E-06,&
9.814E-11,1.093E-09,3.647E-09,2.261E-08,9.251E-08,9.881E-07,2.856E-05,&
9.126E-10,1.050E-08,3.527E-08,2.139E-07,8.518E-07,7.683E-06,1.389E-04,&
8.010E-09,9.708E-08,3.356E-07,1.982E-06,7.721E-06,6.054E-05,7.818E-04,&
7.815E-08,8.794E-07,3.245E-06,1.816E-05,7.060E-05,4.813E-04,4.708E-03,&
7.461E-07,7.480E-06,2.980E-05,1.487E-04,5.893E-04,3.143E-03,2.591E-02,&
7.537E-06,6.759E-05,2.817E-04,1.210E-03,4.891E-03,1.927E-02,1.611E-01 /
```

**Listing 31 - DATA Statement Excerpt**

describe the initialization of arrays of data where the DATA statements require many pages to list. For example, the SARB library includes a module named SKTBL\_02a.f90 that includes 12,886 lines of DATA statements.

Listing 31 is an excerpt from SKTBL\_02a.f90 that illustrates the density of the data. When I scroll through data like this for page after page after page, questions start coming to mind. How is this data maintained? How would one update this data with new values? Is this data documented anywhere? Can this data be verified? If we look at the header comments for the owner module, none of these questions are answered.

One could argue that the 12,886 lines of DATA statements are in compliance with the FORTRAN standard because they are defined in a MODULE without the use of a COMMON block<sup>117</sup>. I am asserting that data in this quantity should probably be stored in a file that is a documented part of the SARB subsystem.

### 7.1.2 Follow Good Programming Practices

I can't list all the good programming practices here, and over my programming career I have violated quite a few. I do believe that anyone developing software should adopt and perfect a programming style that incorporates "good practices". There are many software engineering textbooks in print that will tell you more than you ever wanted to know about good programming practices. I am going to zero in on a specific example to illustrate the trouble<sup>118</sup> it caused. Listing 32 is a code excerpt that incorporates array references using array indices that are computed as opposed to iterated via references to a DO loop index. There is nothing wrong with the practice of computing array indices. It is better practice to implicitly control the range of the computed array indices using DO loop limits that enforce the declared array boundaries. In this way no extra code has to be added to make

```
!
! *** Merge data at cloud heights with data in base profiles.
!
      I = 1
      J = 1
      NewLev_Cnt = 0
!
      DO WHILE ( J <= NumLev_Cld)
!
          TempProl_Minus = ModLev ( IP, I) - Delta_CldPro
          TempProl_Plus  = ModLev ( IP, I) + Delta_CldPro
          TempPro2_Minus = ModLev ( IP, I+1) - Delta_CldPro
!
          IF ( TempProl_Plus >= CPres ( J) .and.
              TempProl_Minus <= CPres ( J)) THEN
!
              *** Cloud height coincides with a fixed level. Replace
              fixed level with cloud height.
!
              NewLev_Cnt = NewLev_Cnt + 1
!
              IF ( NewLev_Cnt > 1) THEN
!
                  *** Check for two cloud heights too close together
!
                  PrevCld      = ModLev ( IP, NewLev_Idx ( NewLev_Cnt-1))
                  CurrCld_low  = CPres ( J) + Delta_CldPro
                  CurrCld_high = CPres ( J) - Delta_CldPro
!
                  IF (PrevCld <= CurrCld_low .AND. PrevCld >= CurrCld_high) THEN
!
                      *** Current and previous cloud heights coincide. Do not
                      add current cloud height to vertical profile.
!
                      NewLev_Cnt = NewLev_Cnt - 1
!
              END IF
!
          END IF
!
      END DO
```

<sup>117</sup> This is an instance of the SARB Mystery Anomaly (see 6.3.2.7 SARB Mystery Anomaly) where all the numbers defined in the DATA statements were referenced as zeroes. The only way I could get it to work was to move the DATA statements into the referencing module and put the respective arrays in a named COMMON block!

<sup>118</sup> In this case, trouble translates to loss of time and therefore loss of money.



```

!
!       ELSE
!       *** Map vertical profile level index into cloud level
!       index and replace fixed level with cloud height
!
!       NewLev_Idx ( NewLev_Cnt) = I
!       ModLev ( IP, I) = CPres ( J)
!
!       END IF
!       ELSE
!
!       NewLev_Idx ( NewLev_Cnt) = I
!       ModLev ( IP, I) = CPres ( J)
!
!       END IF
!
!       J = J + 1
!
!       ELSE IF ( TempPro1_Plus <= CPres ( J) .AND.
!               TempPro2_Minus >= CPres ( J) ) THEN
!       *** Add cloud level data to base profiles
!
!       I1 = I + 1
!       I2 = I + 2
!
!       ModLev_Cnt ( IP) = ModLev_Cnt ( IP) + 1
!
!       NLev = ModLev_Cnt ( IP)
!       NMin1 = ModLev_Cnt ( IP) - 1
!       NPlus1 = ModLev_Cnt ( IP) + 1
!
!       ModLev ( IP, I2:NLev) = ModLev ( IP, I1:NMin1)
!       ModLev_Pro03 ( IP, I2:NLev) = ModLev_Pro03 ( IP, I1:NMin1)
!       ModLev_SpHum ( IP, I2:NLev) = ModLev_SpHum ( IP, I1:NMin1)
!       ModLev_Temp ( IP, I2:NLev) = ModLev_Temp ( IP, I1:NMin1)
!       ModLev ( IP, I1) = CPres ( J)
!
!       CALL MOA_Inter (ModLev ( IP, I), ModLev ( IP, I2),
!                       ModLev ( IP, I1),
!                       ModLev_Temp ( IP, I), ModLev_Temp ( IP, I2),
!                       ModLev_SpHum ( IP, I), ModLev_SpHum ( IP, I2),
!                       ModLev_Pro03 ( IP, I), ModLev_Pro03 ( IP, I2),
!                       ModLev_Temp ( IP, I1), ModLev_SpHum ( IP, I1),
!                       ModLev_Pro03 ( IP, I1))
!
!       NewLev_Cnt = NewLev_Cnt + 1
!
!       NewLev_Idx ( NewLev_Cnt) = I1
!
!       J = J + 1
!
!       ELSE
!       *** Current cloud pressure is not to be inserted into profile yet
!
!       I = I + 1
!
!       END IF
!
!       END DO

```

Listing 32 - Code Excerpt with Computed Array Indices

sure the arrays are not accessed outside their boundaries. But this practice is not always an option because we do not always access an array in a sequential fashion and/or we do not always know what the upper or lower boundaries are. In the latter case it is an error to access an array with a computed index without the knowledge of the array boundaries. An inspection of Listing 32 reveals that most of the code is contained within a DO WHILE block that terminates when variable J is less than or equal to the variable, NumLev\_Cld. Further inspection reveals that the DO WHILE block contains only 4 statements, the 4<sup>th</sup> statement being an IF-THEN construct. Looking at the IF statement

we find that it has a TRUE part, an ELSE-IF part, and an ELSE (FALSE) part. Note that both the TRUE part and the ELSE-IF part increment J but the ELSE part does not increment J. Incrementing J is the only way to eventually terminate the DO WHILE loop. Suppose that the IF conditional expression and the ELSE-IF conditional expression never evaluate to TRUE; then J will never get incremented and the DO WHILE loop becomes an infinite loop. Continuing with our supposition, we observe that the only statement in the ELSE part is incrementing the variable I. Given the infinite loop status, variable I will become increasingly large until it rolls over but this is not what actually occurs.

The first time I attempted to execute the SARB Main processor on the Mac G5, another error<sup>119</sup> in the program caused references to the array, Pres\_Pnt, to errantly return zero. Pres\_Pnt is an array of non-zero constant values (PARAMETER) that are used as indices to the arrays ModLev, ModLev\_SpHum, ModLev\_Pro03, and ModLev\_Temp. This other error caused the aforementioned arrays to become ill-defined in a separate procedure. The ModLev array is used to determine what logic path to take in the IF statement in our code excerpt, and the actual errant behavior was for the ELSE part to execute several million times prior to termination of the DO WHILE loop after executing either or both of the TRUE part and the ELSE-IF part. This implies that variable I was attaining values in the millions. Looking back at the TRUE part and the ELSE-IF part we can see that variable I is referenced as a value and it is used directly and indirectly as an array index. Based on the array declarations for the arrays involved, the variable I should never be allowed to exceed the value 29. When I first ran afoul of this problem it was causing segmentation faults on the Mac G5 platform. I corrected this problem by correcting the other problem that caused the array, Pres\_Pnt to appear to be undefined. But this is not a complete fix for the problem. The complete fix should include the modifications that are illustrated in Listing 33.

```

ELSE
  *** Current cloud pressure is not to be inserted into profile yet
  !
  I = I + 1
  IF(I > 29) THEN
    !      REPORT ERROR AND ABORT PROGRAM
    !      END IF
  !
  END IF
!
END DO

```

**Listing 33 - Suggested Fix for Good Programming Practice**

Looking back at the original code excerpt, you can see that there are other computed array indices that depend on variable I. We can minimize the amount of new code that we introduce in this case if we determine the upper boundary for variable I to compensate for the dependent variables I1 and I2. If we could not do this because variable I would be overly restricted, then we would need to introduce checks on the dependent variables as well. In summary, it is good programming practice to write code that self-protects. In this case, it would have prevented a segmentation fault and given us a clue that something else was wrong.

<sup>119</sup> Array Pres\_Pnt was declared and defined without the PARAMETER attribute causing it to appear to be undefined. This behavior is not observed on the SGI platform so it went undetected in that context.

## 7.2 Use Root Sparingly



**Mac says 11 - Use root sparingly!**

No, this is not a reference to a 60's song; I'm referring to the all powerful Unix user named "root" on the Mac G5 platform. I'm bringing this up now because it applies to the example that was introduced in the last section. I was about three weeks late getting started on the SARB conversion project, and I had taken more time than I had initially estimated to convert the PGS Toolkit and CERESlib. I was hoping I could make up the time when I converted the SARB Main Processor.

I installed and built SARBlib and the SARB Main processor, and then I transferred all the necessary input files from the SGI platform to the Mac G5. I was ready to run the first test case for the SARB Main. Previously, I had established the necessary /CERES/sarb directory structure but I had done this using root privilege since I wanted to maintain the SARB directory structure outside of my user space on the Mac G5. In this configuration I was forced to run SARB as root. Big mistake!

The first event to occur was an immediate segmentation fault. Not yet having developed the process to ferret out array boundary violations and invalid floating point operations, I started to debug the Main processor by tracing the logic flow through the program. I was making pretty good progress when I noticed I could no longer print to the network printer. Consequently, I employed the Bill Gates method of correcting the problem by re-booting the G5 only to find that the G5 could no longer boot. By running the SARB Main processor as root, and by repeating the segmentation fault<sup>120</sup> several times, I had destroyed critical parts of the operating system. Because I was operating as root, the corrupted operating system components were updated on the system disk, and consequently when I tried to re-boot, the boot drive was corrupted. I had to re-install the operating system using the Apple Archive/Install mechanism that preserves the user settings. Fortunately, I did not lose the PGS Toolkit and CERESlib installations. Once the Archive/Install was complete, I switched to root and changed the /CERES/sarb directory permissions to make myself the "owner" and the "group" of the sarb directory and all of its subdirectories. I lost two full days while recovering from the damage done by the segmentation fault. Use root sparingly!

## 7.3 Develop a Contingency Plan



**Mac says 12 - I did not anticipate that I would suffer weeks of down time due to hardware problems.**

<sup>120</sup> The segmentation fault was occurring when the operating system detected an attempt to write outside the memory boundaries of the Mac G5. Before the fault was detected the SARB code had been writing garbage all over the system area.

As I mentioned in the last section, I got a late start and I spent more time than I had estimated for installing the PGS Toolkit and CERESlib. Then I destroyed the Darwin operating system (see 7.2 Use Root Sparingly) and lost two more days on my schedule. All was not lost because I had built contingency time into the schedule, and I had plenty of time left in the contingency bank. After all, what else could go wrong?

Following the recovery of the Darwin operating system, I continued to troubleshoot the SARB Main processor on the Mac G5. I corrected the error that caused the segmentation fault, and I was ready for the next problem. It was about 5PM and I started the SARB Main processor. I watched the SARB Main run for over an hour, and I left for the day figuring that I would pick up the troubleshooting activity the next morning. On the following day when I arrived the Mac G5 power was off, and I found myself wondering if anyone had been in my office playing with the G5. This was not the case. The G5 would not power up, and this is when I decided that SARB was a “Mac killer”.

Fortunately, the repair shop only took 24 hours to replace one of the two CPU’s and I was back in business with only the loss of two more days. When I got the Mac G5 back from the repair shop, I resumed troubleshooting of the SARB Main. There were lots of problems to find and correct, and they are described elsewhere in this document. The troubleshooting process was greatly hampered by the Mac G5 experiencing system lockups or freezes from which the only way to recover was by cycling system power. I struggled with this condition for several weeks before I realized that the Mac G5 needed more repairs, and back to the repair shop went the Mac. The repair shop technician used the SARB software to force the system lockup problem to recur, and the result was the replacement of the second CPU. The repair process took two weeks, and by that time my contingency time was all but used up. It was at this juncture that I determined a second target platform would be a highly desirable resource. I ordered a second Mac G5.

Since the second Mac G5 arrived, there have been two additional system crashes<sup>121</sup> on the original Mac G5, and I have been able to recover gracefully by switching over to the backup G5. Unfortunately, the second Mac G5 suffered a hard disk crash and went to the repair shop for a two-week stay. At the time of the crash I was using the second Mac G5 to support the documentation effort, and I was using the original to complete the SARB conversion work. Thus, no time was lost as the original Mac G5 became the backup during the time that the second G5 was in the repair shop.

I thought I had a pretty good estimate for contingencies but I must confess that the estimate was based on potential software issues. In hindsight, I think the original estimated contingency time was about right given the delays that were suffered due to extended troubleshooting. But I did not anticipate that I would suffer weeks of down time due to hardware problems. I also did not anticipate that the DAAC hardware would not be available throughout the period of performance of the conversion effort. This latter case did not affect the conversion effort other than it has delayed the acquisition of the timing statistics for verification of the 10x processing goal.

---

<sup>121</sup> No, I was not running as root and in one of the cases I am pretty sure that the automatic software update process was the cause of the crash.

## 7.4 Eliminate Invalid Floating Point Operations



**Mac says 13 - The conversion process would be much simpler and faster if a comprehensive effort to eliminate invalid floating point operations is conducted on the source platform prior to the conversion effort.**

When I developed the SARB Conversion Plan and when I estimated the time required to perform the conversion, I was working with the erroneous assumption that the SARB source code was free of invalid floating point operations. I did assume that there would be problems associated with floating point operations, and a lot of the estimated contingency time was set aside for this issue. Fortunately, the IBM XLF compiler provides the mechanisms that I documented in section 3.5.4 Complete Build and Sweep for Invalid Floating-Point Operations.

Looking back on the SARB conversion experience I roughly estimate that about 30% of my time was spent tracking down NaN's and INF's due to divide by zero, multiplication of the default large number by almost any other value greater than one, and by invalid usage of intrinsic functions. Since the SGI and the Mac G5 did not necessarily agree on every NaN and INF, the regular occurrence of invalid floating point operations caused the result comparison process to be difficult at best.

I determined that it is possible to sweep for invalid floating point operations for a given data set over a period of a few days<sup>122</sup>. The conversion process would be much simpler and faster if a comprehensive effort to eliminate invalid floating point operations is conducted on the source platform prior to the conversion effort.

## 7.5 Eliminate Array Boundary Violations



**Mac says 14 - The conversion process would be much simpler and faster if a comprehensive effort to eliminate array boundary violations is conducted on the source platform prior to the conversion effort.**

When I developed the SARB Conversion Plan and when I estimated the time required to perform the conversion, I was working with the erroneous assumption that the SARB source code did not access arrays outside their declared boundaries. Fortunately, the IBM XLF compiler provides the mechanisms that I documented in section 3.5.3 Complete Build and Sweep for Array Bounds Violations.

Looking back on the SARB conversion experience I roughly estimate that about 20% of my time was spent tracking down array boundary violations. Since the SGI and the Mac G5 did not react to array boundary violations in the same way, this problem caused the conversion effort to be more difficult.

<sup>122</sup> This is based on experience with the SARB Main processor using three hourly data-cases.

I determined that it is possible to sweep for array bounds violations for a given data set over a period of a few days<sup>123</sup>. The conversion process would be much simpler and faster if a comprehensive effort to eliminate array boundary violations is conducted on the source platform prior to the conversion effort.

## 7.6 Eliminate Mixed Mode Arithmetic



**Mac says 15 - There is a global precision problem that can be minimized by eliminating the introduction of single precision operands into double precision equations.**

In the last two sections I started off by declaring that I did not anticipate invalid floating point operations and array boundary violations, respectively. Based on my previous experience of converting the ERBE science code from CDC mainframe platforms to SUN Sparc-2 platforms, I can not honestly declare that I did not anticipate that I would find mixed mode arithmetic. In the case of the SARB conversion, I believe that there is a global precision problem that can be minimized by eliminating the introduction of single precision operands into double precision equations throughout the SARB library source code.

See section 0 for a real life example of mixed mode arithmetic in the SARB library code. The example in section 0 was taken without modification from subroutine gwtsa\_add in module seiji\_twostreamsolv\_sw\_v20.f. Subroutine gwtsa\_add is a critical contributor to the calculation of short wave flux values. In section 6.3.3.2 Verification Results for the SARB Main Processor, I discussed the problem of Bad Tuned Cloud Fractions (BTCFs) and how I believe that BTCF's are adversely effected by a precision problem in the flux calculations. It is also true that the BTCF's only occur in daytime footprints where the short wave data is sampled. I would be very interested in re-visiting the BCTF problem following the complete elimination of mixed mode arithmetic from the SARB library software.

## 7.7 Consider Developing a Test Suite



**Mac says 16 - If the test suite software indicates a failure, the individual test module should be "smart" enough to indicate what the problem is at least down to the module level if not the subroutine level.**

But SARB already has a test suite! No, SARB has a comparison suite that is designed to compare output files from two different runs on the same input data. The comparison suite programs are excellent for regression testing, and they are very useful for comparing output files from the benchmark platform with the output files from the target platform during a port or a conversion. An exception to this is the comparison software employed

<sup>123</sup> This is based on experience with the SARB Main processor using three hourly data-cases. And, yes, I think array boundary violations are data dependent.

by CER5.0P1. This software informs you of a data mismatch but it does not do anything to help you determine which of a possible 744 files contains the mismatch error (see 6.3.3.1 Verification Results for the SARB Monthly Preprocessors).

Ideally, a test suite comes bundled with the application source code (PGE), and the test suite software is compiled and linked as a part of the initial build process. Upon a successful build of the application software, the test suite software can be optionally invoked. If the test suite software runs successfully, then there is good confidence that a data case can be tested. If the test suite software indicates a failure, the individual test module should be “smart” enough to indicate what the problem is at least down to the module level if not the subroutine level. The test suite test member should also serve as a standalone debugging tool to facilitate troubleshooting.

It only makes sense to develop a robust test suite if the application software will be ported to several different platforms, or if you are going to turn your code over to a third party for use on their to be determined platform. If the application software will live and die on one platform, then it may be cost prohibitive to spend the time and effort to develop a comprehensive test suite. Also, the best time to develop a robust test suite is during the initial development of the application software since the test suite software is immediately useful for debugging and integration activities.

## **7.8 Do Not Abuse the `-qsave` Compiler Switch**



**Mac says 17 - For conversion testing, `-qsave` should be avoided unless it is temporarily desirable to force all local variables to be stored as STATIC objects.**

During the installation of CERESlib I encountered a segmentation fault when I tried to run the CERESlib test suite software. Instead of tracking the problem down I bypassed it by setting the IBM XLF compiler switch, `-qsave` (see page 62). A side effect of `-qsave` is to subvert the effect of `-qinitauto` (see page 20). I continued to use `-qsave` when I moved on to the SARBlib and SARB Main initial tests. In doing so, I unintentionally missed detecting an uninitialized local variable (see section 0 for an explanation) until fairly late in the conversion process. For conversion testing, `-qsave` should be avoided unless it is temporarily desirable to force all local variables to be stored as STATIC objects.

## **7.9 DATA Statements in Module Headers**



**Mac says 18 - The IBM XLF compiler had trouble with global variables declared in the declaration part of a module when DATA statements were used to initialize the variables.**

As documented in section 6.3.2.7 SARB Mystery Anomaly, the IBM XLF compiler had trouble with global variables declared in the declaration part of a module when DATA statements were used to initialize the variables. Fortunately, there were not many instances of this anomaly. It happened frequently enough during the SARB conversion

that I found it necessary to manually scan all the modules in SARBlib for arrays that were initialized with DATA statements. I have not been able to emulate the problem, and I have not seen the problem documented in IBM's known problem list for the XLF compiler. Therefore, I would be looking for it in any future conversion efforts.



## 8 Writing Optimal Code

I've heard people say that with today's technology, nearly unlimited physical memory, and astronomical CPU speeds, programmers don't have to be so conscientious about keeping their code smaller and faster. I often ask myself if the programmers at Microsoft subscribe to that philosophy but that's another story. I do not agree with the idea that technology makes up for a programmer's ineptness; rather, I try to challenge myself to find the most elegant solution to the problem. Sometimes there is not enough time to continue iterating on software that may already be working correctly, so I try to get the first version as clean as possible because I may not get another chance to clean it up later on.

This conversion guide is not a tutorial on programming practices so I am only going to touch on some of the subjects that I think have the most benefit to those programmers who may find themselves writing subsystem code or modifying subsystem code that was written by someone else, possibly on another planet<sup>124</sup>. Sometimes, when you inherit someone else's code, you are not so much concerned with how fast the code executes but rather by how difficult it is to follow the logic flow. For example, can you tell where a block structure ends, or do all the loops and if statements seem to blend into a complicated blob? Does the procedure logic flow from top to bottom or does it bounce around? Are there comments, and if there are comments are they relevant?

### 8.1 Comment Your Code

How does commenting your code make the code optimal? Technically speaking, commenting source code does not make it optimal. From a maintenance perspective, well commented code might very well be optimal code in that it might take just a few minutes to locate the best place to make a change or to isolate the most probable point of failure. This would be in contrast to searching for a subroutine that may be a target for change or the possible site of a logic error. You open the subroutine in your favorite editor and all you see are cryptic variable names and line after line of assignment statements with complicated right-hand-sides that are perhaps continued for several lines. Listing 34 contains a code excerpt that incorporates very few comments. The code does indent the DO-loop blocks but I would argue that the indenting style adds to the chaotic nature of this code. There are four comments that apparently identify the date of the commented code; perhaps this was an update. The "7" and the "11" in parentheses have no meaning to me; perhaps this commented code was taken from some software that had numbered sources like a bibliography. However, after I stared at the code that is surrounded by comments I decided that they are most likely bug fixes; in one case a divide by zero and in the other case an undefined initial value. So, what does this code excerpt do? I can't tell from looking at it, can you? Let's just indulge this example one more step. Examine the following line extracted from the code excerpt in Listing 34:

```
xx = yy * ( 1.0 - eex ) * 2.0 + 6.2831854 * eex * bs
```

<sup>124</sup> If you read much code written by others, sooner or later you will find yourself asking the question, what planet is this person from?

```

do j = 1, nq
  t0 = 0.0
  do i = 2, mdfs
    il = i - 1
    fx(il,j) = exp ( - ( t(il) - t0 ) / ug(j) )
    fy(il) = expn(il)
    xx = lamdan(il) * ug(j)
    fz1(il,j) = ( 1.0 - fx(il,j) * fy(il) ) / ( xx + 1.0 )
    fz2(il,j) = ( fx(il,j) - fy(il) ) / ( xx - 1.0 )
    ub(il,j) = ug(j) * beta(il)

c----- 4/2/97 (7)
    if (ub(il,j) .eq. 1.0) ub(il,j) = 1.001
c----- 4/2/97 (7)

    fid(i,j) = fid(il,j) * fx(il,j) + fj(il) * fz1(il,j) +
1          fk(il) * fz2(il,j) +
1          fuq2(il) / ( ub(il,j) + 1.0 ) *
1          ( alfa(i) - alfa(il) * fx(il,j) )

    t0 = t(il)
  enddo
enddo

yy = 0.0
do j = 1, nq
  yy = yy + ugw(j) * fid(mdfs,j)
enddo

xx = yy * ( 1.0 - eex ) * 2.0 + 6.2831854 * eex * bs
do j = 1, nq
  fiu(mdfs,j) = xx
enddo
c 6-24-98 (11)
fiur(mdfs) = xx
c 6-24-98 (11)
do j = 1, nq
  do i = mdfs - 1, 1, -1
    fiu(i,j) = fiu(i+1,j) * fx(i,j) + fg(i) * fz2(i,j) +
1          fh(i) * fz1(i,j) +
1          fuq1(i) / ( ub(i,j) - 1.0 ) *
1          ( alfa(i+1) * fx(i,j) - alfa(i) )

    enddo
  enddo

```

Listing 34 - Code Excerpt With No Comments

Is that constant, 6.2831854, supposed to be  $2\pi$ ? Last time I checked  $2\pi$  was 6.283185307..., so maybe it's not supposed to be  $2\pi$ . Of course, I am quibbling about the 8<sup>th</sup> digit of single precision significance so it does not make any difference does it? Also, I thought the CERES library defined  $\pi$  so that everyone who wrote subsystem code would use the same constants. This intrigued me so I did a grep on the source directory for SARBlib and obtained the results illustrated in Listing 35. I am starting to digress from my original point but in every case of the variances from the CERESlib definition, I think there should be an inline comment explaining why<sup>125</sup>.

How should code be commented? The algorithm defined by the code should be briefly included and interspersed with the code using English language comments. There should be at least one meaningful comment line for every three or four lines of code.

<sup>125</sup> If you take a careful look at the grep example you will see that in module `seiji_twostreamsolv_sw_v20.f` poor old pi is being defined out to 10 places in a single precision format. This begs the question, did the computation at hand need those last 3 digits of precision or not?

Commenting is not hard; it just requires discipline and a sense of respect for the people who may have to work with the code in the future.

```
[CTS1-105:sarb/lib/src] Donaldsn% grep "3.1415" .f
WindowFilter.f: adm(ib) = fu_sf(ib) /(3.14159*fu_sr(ib) )
WindowFilter.f: sat_f   = sat_f   + sat_unf_sr(ib) * (3.14159*adm(ib))
aqua_wnflt_0404.f:   adm(ib) = fu_sf(ib) /(3.14159*fu_sr(ib) )
aqua_wnflt_0404.f:   sat_f   = sat_f   + sat_unf_sr(ib) * (3.14159*adm(ib))
fuprint.f90:pi=3.14159
misc_0403.f:   fw3 = u0 * 3.14159 * f0
misc_0403.f:   fw = 3.1415927 * f0 * w * exp ( - fq * t0 )
misc_0403.f:   fw = 3.1415927 * f0
misc_0403.f:   ssfc = 3.1415927 * u0q(1) * exp(-t(ndfs)/u0q(1)) * rsfc * f0(1)
misc_0403.f:   ssfc = 3.1415927 * asbs
misc_0403.f:   fw3 = u0 * 3.1415927 * f0
misc_0403.f:   data pi /3.1415927/
rad_multi_0403.f90:f0 = 1.0 / 3.14159
sei_ji_twostreamsolv_sw_v20.f:   data pi / 3.141592654 /
sei_ji_twostreamsolv_sw_v20.f:   data pi / 3.14159265 /
sfcalb_history.f90:   dx=radiuskm/( 111.* cos(rlat*3.14159/180.) )
uvcor_all.f90:   val(1) = acos(uvco%u0) *(180.0/3.14159) ! Cos Sol to SZA
uvcor_all.f90:   val(1) = acos(uvco%u0) *(180.0/3.14159) ! Cos Sol to SZA
uvcor_all.f90:   val(2) = acos(uvco%u0) *(180.0/3.14159) ! Cos Sol to SZA
uvcor_all.f90:   val(2) = acos(uvco%u0) *(180.0/3.14159) ! Cos Sol to SZA
```

**Listing 35 - grep Results for 3.1415**

## 8.2 Source-Level Optimization

I have heard the argument that a programmer should not waste time optimizing source code because the compiler will do it at compile time. I think this statement is mostly true. I think that if the programmer spends some extra time optimizing the source code, the compiler may be able to do a better optimization at compile time, or the compiler might be able to perform a higher level of optimization. I am going to discuss a few kinds of source-level optimization that may help the compiler do its job.

### 8.2.1 Use Array Notation Instead of Pointers

This is not a lesson learned from SARB. As far as I can tell SARB does not use pointers. The use of pointers makes it more difficult for any compiler to optimize the code. Using array notation makes the task of the optimizer easier by reducing possible aliasing. To optimize pointer code, the compiler has to perform detailed and aggressive pointer analysis. For example, any number of pointers can be set up to point to the same memory location but using array notation this is not the case.

### 8.2.2 Unrolling Small Loops

This one goes back to the argument that you should let the compiler do the work for you. I agree except for the case where we have a small loop that is being initialized and executed several 10's of millions of times. For example, the SARB main processor processes on the order of 100,000 footprints for one hour of data. During the processing of a single footprint there is a logic path that is executed iteratively. Suppose that somewhere along the logic path there is a small loop that is embedded within two outer loops and as a result the small loop is initialized and executed 1000 times. Over the course of 100,000 footprints the statements in our hypothetical loop will each execute 100,000,000 times making this particular loop a candidate for loop unrolling. For example, consider the loop in Listing 36.

```

!      3D-transform: Multiply vector V by 4x4 transform matrix M
      DO I = 1, 4
        R(I) = 0.0
        DO J = 1, 4
          R(I) = R(I) + M(J, I) * V(J)
        END DO
      END DO

```

**Listing 36 - Example for Loop Unrolling**

Replace this code with the code illustrated in Listing 37.

```

!      3D-transform: Multiply vector V by 4x4 transform matrix M
      R(1) = M(1, 1) * V(1) + M(2, 1) * V(2) + M(3, 1) * V(3) + M(4, 1) * V(4)
      R(2) = M(1, 2) * V(1) + M(2, 2) * V(2) + M(3, 2) * V(3) + M(4, 2) * V(4)
      R(3) = M(1, 3) * V(1) + M(2, 3) * V(2) + M(3, 3) * V(3) + M(4, 3) * V(4)
      R(4) = M(1, 4) * V(1) + M(2, 4) * V(2) + M(3, 4) * V(3) + M(4, 4) * V(4)

```

**Listing 37 - Results of Loop Unroll**

Once again, we would only do this if the target loop is small and it executes millions of times. In this particular example, the savings would come from eliminating the overhead processing for the two DO-loops.

### 8.2.3 Long Logical IF Expressions

It is a good idea to get in the habit of designing IF conditional expressions to avoid the FALSE branches. For example, the following conditional expression is preferred if most of the data tested is within range:

```
IF(A <= MAX .AND. A >= MIN .AND. B <= MAX .AND. B >= MIN)
```

If most of the data is out of range then the following is preferred:

```
IF(A > MAX .OR. A < MIN .OR. B > MAX .OR. B < MIN)
```

The intent is to minimize the number of times that the conditional expression evaluates to FALSE causing a branch around the TRUE part.

### 8.2.4 Arrange Boolean Operands for Quick Expression Evaluation

In expressions that use the logical AND or logical OR operator, arrange the operands for quick evaluation of the expression to exploit the ability of the compiler to short-circuit the expression evaluation. For example, in an expression that uses the logical AND operator, the first operand to evaluate to FALSE will terminate the evaluation and subsequent operands do not have to be evaluated. In an expression that uses the logical OR operator, the first operand to evaluate to TRUE will terminate the evaluation.

### 8.2.5 Unnecessary Store-to-Load Dependencies

A store-to-load dependency exists when data is stored to memory only to be read back shortly thereafter. Many compilers can optimize store-to-load dependencies but if the dependency occurs while operating on arrays, the compiler may not be able to perform the optimization. The programmer can help the compiler by removing store-to-load dependencies manually. For example, by introducing a temporary variable that the compiler can allocate to a register a significant performance increase can be gained.

```

DOUBLE PRECISION x(VECLEN), y(VECLEN), z(VECLEN)
INTEGER K

DO K = 2, VECLEN
  x(K) = x(K - 1) + y(K)
END DO

DO K = 2, VECLEN
  x(K) = z(K) * (y(K) - x(K - 1))
END DO

```

**Listing 38 - Store-to-Load Dependency**

Listing 38 illustrates code with a store-to-load dependency.

```

DOUBLE PRECISION x(VECLEN), y(VECLEN), z(VECLEN)
INTEGER K
DOUBLE PRECISION t

t = x(1)
DO K = 2, VECLEN
  t = t + y(K)
  x(K) = t
END DO

t = x(1)
DO K = 2, VECLEN
  t = z(K) * (y(K) - t)
  x(K) = t
END DO

```

**Listing 39 - Avoiding Store-to-Load Dependency**

Listing 39 illustrates the modified code with the addition of a temporary variable used to eliminate the store-to-load dependency at the source level.

### 8.2.6 Arranging Cases by Probability of Occurrence

Arrange CASE statement cases by probability of occurrence from most probable to least probable.

```

SELECT CASE (days_in_month)
  CASE(28:29) short_months = short_months + 1
  CASE(30)    normal_months = normal_months + 1
  CASE(31)    long_months = long_months + 1
  CASE DEFAULT
    Print*, 'Days in month outside the range [28, 31]'
END SELECT

```

**Listing 40 - CASE Statement Not in Most Probable Order**

If the compiler translates the CASE statement to a comparison chain, then an improvement can be gained. If the compiler translates the CASE statement to a jump table, then the arrangement of the cases in most probable order will have no negative impact. For example, the CASE statement in Listing 40 is not in most probable order.

Listing 41 is the same CASE statement in most probable order.

```
SELECT CASE (days_in_month)
  CASE(31)    long_months = long_months + 1
  CASE(30)    normal_months = normal_months + 1
  CASE(28:29) short_months = short_months + 1
  CASE DEFAULT
    Print*, 'Days in month outside the range [28, 31]'
END SELECT
```

**Listing 41 - CASE Statement in Most Probable Order**

### 8.2.7 Generic Loop Hoisting

Reduce redundant constant expression evaluation to improve the performance of inner loops. For example, the code in Listing 42 should be avoided.

```
DO I = 1, upper_bound
  IF(CONSTANT > 100) THEN
    CALL GreaterThan_100(I)
  ELSE
    CALL LessThanOrEqual_100(I)
  END IF
END DO
```

**Listing 42 - Redundant Constant Evaluation Code**

The preferred optimization is illustrated in Listing 43.

```
IF(CONSTANT > 100) THEN
  DO I = 1, upper_bound
    CALL GreaterThan_100(I)
  END DO
ELSE
  DO I = 1, upper_bound
    CALL LessThanOrEqual_100(I)
  END DO
END IF
```

**Listing 43 - Reduction of Loop-invariant Constant Expression**

### 8.2.8 Sorting and Padding User Defined Types

Help the compiler to optimize user defined type variable access by sorting and padding the variable declarations. This may only apply to user defined type declarations that include the SEQUENCE statement but it is a good habit to incorporate in your coding style. Follow these steps to sort and pad your user defined types:

- Sort the derived type members according to their type sizes, declaring members with larger type sizes ahead of members with smaller type sizes
- If possible pad the user defined type such that the overall size is a multiple of the largest members type size

## 9 Writing Production Code

I have agonized greatly over what I would say in this section. I could have written a how-to approach to writing production code but I think it would be redundant having just finished the last two sections on lessons learned and writing optimal code. Suffice it to say that if the code that you deliver for production is running efficiently, generating valid results, and meeting the scientific objectives for the current release, then the code is ready for the production environment. Also, it may be unfair to demand that the CERES subsystems meet production code readiness standards when most of the subsystems are continually being updated with new code, or they are modifying older code. Thus, we may be asking too much to move code from development to production when the latest round of updates compiles and executes without a runtime error. But wait, my last statement does not meet the standard for production readiness<sup>126</sup>, does it? An important step has been left out. It is critical that we test the development code to 1) verify that it still meets the scientific objectives that were established prior to the latest round of modifications (regression testing), 2) verify that it correctly implements the new requirements (implies developing test modules), and 3) verify that the resultant program is still running efficiently (implies a benchmark set of defined performance metrics)<sup>127</sup>. If we do not take the time to verify our work prior to producing data products, then we are vulnerable to loss of credibility with our clients, our peers, and our competitors.

---

<sup>126</sup> What is the CERES standard for production readiness?

<sup>127</sup> This is less than a minimum set. It is assumed that the code developers are familiar with and actively using modern software testing techniques that are the subject of any beginning software engineering course.

## 10 Summary

In preparation for the migration of the CERES science code to commodity-based open source platforms, a pilot project was undertaken to test the feasibility of maintaining 10x processing on the new platform architecture, and to be a pathfinder for the software conversion process for the CERES subsystem software developers. The pilot project involved the complete conversion of the CERES SARB subsystem from the SGI platform to an IBM PowerPC-970 based platform. The pilot project also included a port of the SARB software from the PowerPC-970 development platform to a PowerPC-970 based cluster in preparation for full scale SARB production. Unfortunately, the port to the cluster has not occurred because the new cluster is not yet available for testing at this writing.

A conversion plan was written and delivered to the CERES Data Manager along with a comprehensive schedule for one person working at a 70% level of effort for a period of one year. The SARB conversion was completed within the schedule and this document was created to document the lessons learned and to describe a process that can be generically applied to the remaining CERES subsystems. This document goes beyond the generic process of converting CERES science code in that it also documents the IBM XLF compiler and some of the other useful resources available on the proposed development platform, the Apple Macintosh dual processor G5 computer. This document also reports the findings from the PGS Toolkit conversion, the CERES Library conversion, and the SARB subsystem conversion process.

Although the cluster testing has not yet been conducted, the wall clock timing comparisons between the SGI platform and the Mac G5 for SARB were very promising giving way to optimism that the SGI performance will be surpassed when the new cluster is ready for production. It is estimated that the cluster will be available within two months following this writing, and the converted SARB code will be ported and tested for 10x processing feasibility. Also, a more succinct generic cookbook conversion process will be distilled from this document.